Software defined radio design for an IEEE 802.11A transceiver using open source Software Communications Architecture (SCA) implementa

Leong, Wai Kiat Chris

Monterey California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SOFTWARE DEFINED RADIO DESIGN FOR AN IEEE 802.11A TRANSCEIVER USING OPEN SOURCE SOFTWARE COMMUNICATIONS ARCHITECTURE (SCA) IMPLEMENTATION::EMBEDDED (OSSIE)

by

Leong Wai Kiat Chris

December 2006

Thesis Advisor:	Frank Kragh
Thesis 2nd Reader:	R. Clark Robertson

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Software Defined Radio design for An IEEE 802.11a Transceiver using Open Source Software Communications Architecture (SCA) Implementation::Embedded (OSSIE)		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Leong Wai Kiat Chris			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

In this thesis, we present the design of a software defined radio (SDR) transceiver using Open Source SCA Implementation::Embedded (OSSIE) as the software platform. Designing a SDR requires both an appreciation of the IEEE 802.11a (wireless Local Area Network at 5 GHz band) protocol standard as well as the understanding of the C++ and CORBA software tools available to implement the physical transmitter and receiver layers. For this work, the Incremental Development Model was chosen, which is comprised of three stages: Design, Develop and Verify. The advantage of this model is its incremental nature, which allows the developer to learn from earlier versions of the system. Implementing the IEEE 802.11a physical layer using OSSIE requires a total of 23 components, 12 different functionalities and 31 sequential input-output (I/O) processes for the transmitter, while the receiver is implemented with 18 components, 12 different functionalities and 20 sequential I/O processes. The completed transmitter and receiver layers are validated successfully according to test cases stipulated in the IEEE standard.

14. SUBJECT TERMS Software Defined Radio, IEEE 802.11a, wireless Local Area Network, Open Source SCA Implementation::Embedded (OSSIE), C++, CORBA			**15. NUMBER OF PAGES** 138
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

SOFTWARE DEFINED RADIO DESIGN FOR AN IEEE 802.11A TRANSCEIVER USING OPEN SOURCE SOFTWARE COMMUNICATIONS ARCHITECTURE (SCA) IMPLEMENTATION::EMBEDDED (OSSIE)

Leong Wai Kiat Chris
Major, Republic of Singapore Air Force
B.Eng, National University of Singapore, 1999
M.Tech, National University of Singapore, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2006

Author: Leong Wai Kiat Chris

Approved by: Assistant Professor Frank Kragh
Thesis Advisor

Professor R. Clark Robertson
Thesis 2nd Reader

Professor Jeffrey B. Knorr
Chairman, Electrical and Computer Engineering Department

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In this thesis, we present the design of a software defined radio (SDR) transceiver using Open Source Software Communications Architecture (SCA) Implementation::Embedded (OSSIE) as the software platform. Designing a SDR requires both an appreciation of the IEEE 802.11a (wireless Local Area Network at 5 GHz band) protocol standard as well as the understanding of the C++ and CORBA software tools available to implement the physical transmitter and receiver layers. For this work, the Incremental Development Model was chosen, which is comprised of three stages: Design, Develop and Verify. The advantage of this model is its incremental nature, which allows the developer to learn from earlier versions of the system. Implementing the IEEE 802.11a physical layer using OSSIE requires a total of 23 components, 12 different functionalities and 31 sequential input-output (I/O) processes for the transmitter, while the receiver is implemented with 18 components, 12 different functionalities and 20 sequential I/O processes. The completed transmitter and receiver layers are validated successfully according to test cases stipulated in the IEEE standard.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my gratitude to Assistant Professor Frank Kragh and Professor R. Clark Robertson for their professional advice, guidance and assistance in making this thesis a possibility in such a short time. Their patience in me is greatly appreciated. I am also grateful for my organization, the Republic of Singapore Air Force, for giving me this opportunity to study at the Naval Postgraduate School and carry out this interesting thesis work. This one-year experience has definitely enriched my knowledge in my professional and technical fields.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Reed defines a software defined radio (SDR) as a radio that can be "substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software"[1]. SDR has distinct military advantages over conventional radio as it promotes multi-functionality, mobility, compactness, flexibility, ease of manufacture and ease of upgrades.

A military unit will not always know in advance what communications capabilities it will need in operations. This is especially true in coalition operations, where the coalition partner's forces may not have the preferred radio equipment. Therefore, in operations, it is imperative to be prepared for many different means of communications, especially those that a coalition partner would be likely to possess. Radio equipment built to commercial (i.e., IEEE wireless) standards is just such a likely means of communications. SDR with the software to communicate in many modes, including commercial standards, would be a substantial advantage to a military unit that is part of a coalition operation, when time and foresight may not be sufficient for the fielding of communications equipment ideally suited for the specific coalition membership. For this research, the focus is on software design for the commercial standard IEEE 802.11a implemented on a SDR.

In this thesis research, the transceiver components shall be implemented using software radio techniques. The components will be designed for use in an IEEE 802.11a transceiver and for contribution to the library of components being developed. The

[1] J. H. Reed, "Software Radio: A Modern Approach to Radio Engineering", 1st ed. New Jersey: Prentice Hall, 2002.

components developed shall be flexible so that they can be modified to implement other receivers by customizing the appropriate parameters. Design of the SDR shall use the Software Communications Architecture (SCA) including Common Object Request Broker Architecture (CORBA) as dictated for the Joint Tactical Radio System (JTRS). The components shall be tested based on functions and test cases found in the IEEE 802.11a standard.

For the transmitter, all functionalities from the input binary data to the digitized input to the Digital-to-Analog Converter (DAC) will be implemented in software. Similarly for the receiver, all functionalities after the Analog-to-Digital Converter (ADC) to the regeneration of the binary received information will be implemented in this thesis work. It is important to note that all software components are implemented at base band, i.e., before up-conversion at the transmitter and after down-conversion at the receiver.

Following the principle of iterative and incremental development, five models have been developed, with each being more complex and built on the experiences gathered from the previous. The first three are exploratory models using MATLAB, which are relatively easy to build since many of the radio functionalities are already available as function calls. The fourth model builds on the success of the MATLAB design. It emulates a Transmitter-Receiver (Tx-Rx) design using Open Source SCA Implementation::Embedded (OSSIE) but following closely the previous MATLAB model. The final model is the full scale OSSIE implementation of IEEE 802.11a PHY layer, which is the primary objective of this thesis work.

In this thesis, we have successfully met the following objectives:

1. The IEEE 802.11a PHY layer transmitter has been built using a total of 23 OSSIE components with 12 different functionalities and 31 sequential I/O processes. Correspondingly, the receiver is implemented using 18 components with 12 different functionalities and 20 sequential I/O processes.

2. All these components have been designed with modularity and flexibility in mind so that they contribute to the pool of components for future radio design. "*Readme*" files are also included in each component's directory to explain its I/O data types, functionalities and assumptions. Appropriate parameters can be modified easily for use in other transceivers. All the files mentioned in this research have been included in the reference CD.

3. With the design implemented fully in the OSSIE waveform development environment, the SDR conforms to Software Communications Architecture (SCA) and the Common Object Request Broker Architecture (CORBA). This will ensure flexibility, performance and maximum potential for software module reuse.

4. Using the test cases provided in Annex G of the IEEE 802.11a standard document, all the components have been verified to provide the necessary functionalities expected of them.

The software components developed here shall serve as a baseline to link up with other software or hardware components to implement a fully functional IEEE 802.11a transceiver. This functionality then can be added to any SDR that includes the minimum hardware functionality, (i.e. bandwidth, frequency band, sample rates, signal processing complexity) and conforms to the design standards specified by the JTRS JPEO in the

SCA, thereby providing that radio user one more mode of communications which extends readiness to include perhaps unanticipated communications demands.

I. INTRODUCTION

A. OBJECTIVES

In designing the software-defined radio (SDR), the following objectives have been identified:

1. Design and implement transceiver components using soft radio techniques. The components will be designed for use in an IEEE 802.11a transceiver and for contribution to the library of components being developed.

2. The components developed shall be flexible so that they can be modified to implement other receivers by customizing the appropriate parameters.

3. Design of the SDR shall use the Software Communications Architecture (SCA) including Common Object Request Broker Architecture (CORBA) for flexibility, performance and maximum potential for software module reuse.

4. The components shall be tested based on functions and test cases found in the IEEE 802.11a standard.

B. GUIDING PRINCIPLES

Designing a SDR requires both the appreciation of the protocol standard as well as the understanding of the software tools available to implement the physical transmitter and receiver layers (layer 1 under the OSI 7 layers model). In order to implement the coding effectively and efficiently within the limited amount of time, it is important that the whole research should be conducted with a set of guiding principles in mind. The following three are single out as critical factors guiding the research that has been carried out.

1. Start Small

Implementing the 802.11a physical layer using Open Source Software Communications Architecture (SCA) Implementation::Embedded (OSSIE) requires a total of 23 components, 12 different functionalities and 31 sequential input-output (I/O) processes for the transmitter, while the receiver is implemented with 18 components, 12 different functionalities and 20 sequential I/O processes. It would be a daunting task to

jump straight into the coding of a full-scale IEEE 802.11a standard as it is extremely complex and would probably result in a demoralizing outcome.

Hence, the strategy is to 'start small' by first developing simple components that work. This will help to build up confidence and experience in using the OSSIE software, which is still a trial version. This assimilation time is needed to understand the programming language and flow. An Incremental Development Model has been chosen for the software implementation as it advocates the need to be modular and provides constant feedback in the design cycle to minimize back tracing. It minimizes major bugs from occurring in the design further downstream in the implementation. More details on the model are provided in the next section.

2. Think Modular

As this research is more of a discovery venture (since it is the first time an attempt has been made to use OSSIE to implement IEEE 802.11a standards), the push for a direct working design outweighs the need for an efficient one. Hence, it is more important to get the various components under the standard to carry out their necessary functions, even though the code may not be written as efficiently as desired. If there is a need, future efforts can be recommended to optimize the code and integrate it with other aspects of the standards or hardware. These further enhancements are proposed in the concluding chapter.

The targets to be modular and reusable reinforce the need to keep the components 'simple' so that they can be understood and modified easily for future enhancement. While the OSSIE waveform developer already provides handy tools to modify components, it is critical to have good programming discipline in managing the complexity of the software algorithm. This prevents the code from getting too exclusive and losing the flexibility of customization.

3. Help is Out There

As mentioned before, OSSIE is still under development and refinement. It is very important that one is kept up to date regarding the OSSIE software development to fully

utilize its capabilities. Through the research, we have been fortunate to have constant dialogue and guidance from the OSSIE development team at the Virginia Polytechnic Institute and State University (Virginia Tech).

The algorithm, functions and objects in the software are written in the C++ programming language. However, it is equally important to appreciate the underlying CORBA interfaces that enable input/output (I/O) interaction between components and integration of the transmitter and receiver waveforms. Another challenge will be to understand the IEEE 802.11a communication standard (e.g. modulation, error corrections, orthogonal frequency division multiplexing) and convert that into the desired algorithms in the C++ programming language.

To fully understand the various technical details and challenges on one's own is nearly impossible in such a short time. It has been important to seek assistance quickly whenever the implementation reached an obstacle. Proven algorithms and approaches are referenced so as not to reinvent the wheel. This research is also a collaboration with Major Low Kian Wai, who was working on the IEEE 802.16 implementation using OSSIE. Various useful resources include literature studies, Internet research, sample C++ software algorithms, MATLAB simulation for IEEE 802.11a standard, etc. All of these resources come disjointed but they provide guidance and the tools to complete the thesis research.

C. INCREMENTAL DEVELOPMENT MODEL

The intent of this model is to develop a software system incrementally, allowing the developer to take advantage of what has been learned in earlier versions of the system. The process starts with a simple implementation of a subset of the software requirements and iteratively enhances the evolving versions until the full system is implemented. At each iteration, design modifications are introduced and new functional capabilities are included [1]. The incremental development model has three stages: **Design**, **Develop** and **Verify**. Figure 1 describes the interrelationship between these three stages as a model and how it corresponds to processes in the software waveform development of the IEEE 802.11a standard.

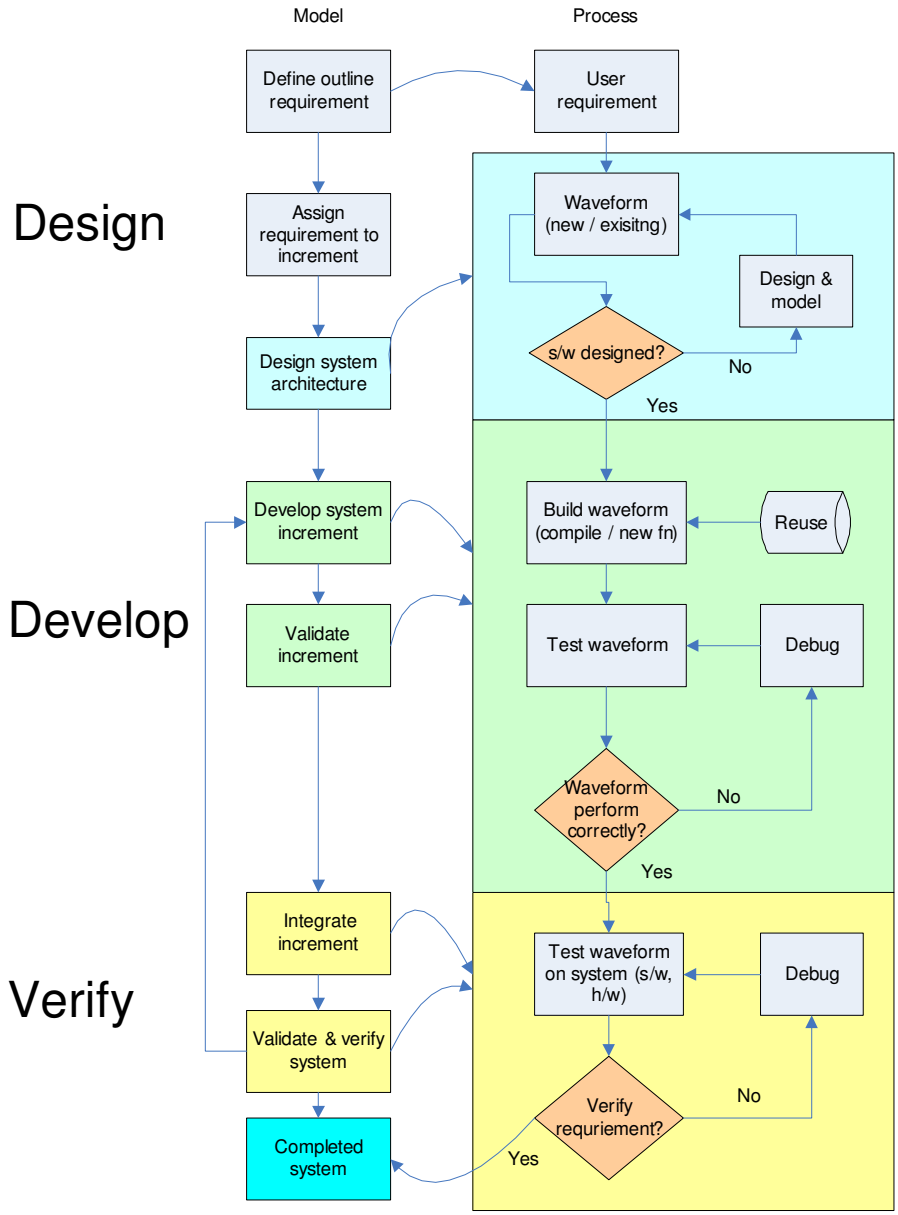

Figure 1. Incremental Development Model.[2]

1. Design

This stage starts with defining the outline software requirements and assigning these requirements to the specific increment. From these requirements, the system architecture is designed to serve as a framework for actual software development in the next stage.

[2] Eric Christensen, Elisa Wing, "Waveform application development process for software defined radios", IEEE article, Motorola SSG and SPAWARSYSCEN, 2000: 234

2. Develop

This is the actual 'hands on' of software development and programming, whereby the system requirements and pseudo-codes are converted to actual software languages. The coded algorithms are validated incrementally to ensure they meet the functionality expectations. Successful increments are stored for future use and new functionalities through design modifications are introduced for the next increment.

3. Verify

With the incremental development, the software system design gets larger and more complex. Increments shall be integrated in this stage and verified that the system as a whole is able to meet the holistic software requirements. For this research, the completed system must be able to emulate the IEEE 802.11a physical layer for both the transmitter and the receiver.

D. THESIS CHAPTERS BREAKDOWN

In this thesis, we present the approach of implementing a SDR transceiver using OSSIE as the software platform. This work has been divided into seven chapters. The following shows the focus of each chapter:

Chapter I: Introduction. This chapter begins by giving an overview of the thesis objectives and follow by describing the guiding principles behind the design. The Incremental Development Model is also discussed to set the framework for subsequent chapters.

Chapter II: Design. In this chapter, the key concepts stipulated in the thesis title are conveyed as requirements. Conceptual design using MATLAB models are discussed before the OSSIE detailed design model is presented. In the detailed design, transmitter and receiver models are discussed separately.

Chapter III: Develop – Transmitter. The design of the transmitter is presented in this chapter. The discussion is divided into preamble, SIGNAL and DATA subframes. It ends with the design of concatenating the three subframes to form the PPDU frame.

Chapter IV: Develop – Receiver. The design of the receiver is presented in this chapter. Similar to the transmitter, the chapter is divided into preamble, SIGNAL and DATA subframes.

Chapter V: Challenges. The IFFT/FFT and Viterbi decoder are singled out as special interest components as their design are both involved and complex. Difficulties and peculiarities of OSSIE software are also mentioned.

Chapter VI: Verify. Testing of the final product is carried out with reference to the IEEE 802.11a standard. The test results for both transmitter and receiver models are presented in this chapter.

Chapter VII: Conclusion. This chapter gives a review on the thesis objectives and provides recommendations on potential future work with the baseline components developed.

II. DESIGN

A. REQUIREMENTS ANALYSIS

In order to meet the thesis objectives, it is important to fully understand the concepts underlying base on the thesis topic – "*Software Defined Radio design for an IEEE 802.11a Transceiver using OSSIE*". The three important concepts are Software Defined Radio, the IEEE 802.11a wireless standard and OSSIE. In the following section, these three concepts are presented in sufficient detail to set the design boundaries. This shall lead to the conceptual design of the eventual software architecture.

1. Software Defined Radio

SDR refers to a radio that can be "substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software" [2]. SDR has advantages over conventional radio as it promotes multi-functionality, mobility, compactness, ease of manufacture and ease of upgrades. The design of a SDR generally comprises a series of procedures that include system engineering, RF chain planning, Analog-to-Digital and Digital-to-Analog hardware selections, software and hardware architecture selection and radio validations. For this research, the focus is only on software architecting according to a specific standard – IEEE 802.11a.

The extent of software architecting or the boundary where software algorithms shall be written is shown in Figure 2.

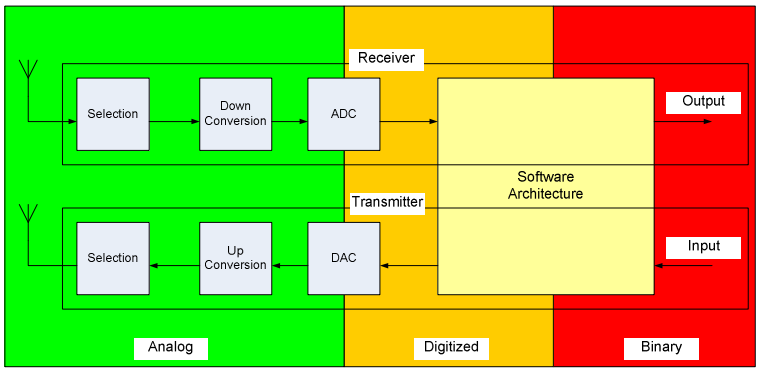

Figure 2. Model of Software Defined Radio.

For the transmitter, all functionalities from the input binary data to the digitized input to the Digital-to-Analog Converter (DAC) will be implemented in software. Similarly, for the receiver, all functionalities after the Analog-to-Digital Converter

(ADC) to the regeneration of the binary received information will be implemented in this thesis work. It is important to note that all software components are implemented at base band, i.e., before up-conversion at the transmitter and after down-conversion at the receiver.

2. IEEE 802.11a PHY Layer

The physical standard takes reference from Part11: IEEE Std 802.11a-1999 (Revision 2003) [3]. It describes the wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications, specifically for high-speed physical layer in the 5 GHz band. Since the following implementation is done at base band, the carrier frequency of approximately 5 GHz band is immaterial. IEEE 802.11a is based on Orthogonal Frequency-Division Multiplexing (OFDM) whereby a single transmission is encoded into multiple subcarriers. Section 17 of the standard (OFDM PHY specification for the 5 GHz band) is the working document upon which this thesis's algorithm is based. A simplified explanation of the working of the OFDM PHY layer can also be found in reference [4].

Important design requirements of an IEEE 802.11a PHY system are as follows:

a) data payload communication capabilities of 6, 9, 12, 18, 24, 36, 48, and 54 Mbits/s

b) mandatory transmitting and receiving at data rates of 6, 12, and 24 Mbits/s

c) 52 subcarriers that are modulated using binary or quadrature phase shift keying (BPSK/QPSK), 16-quadrature amplitude modulation (QAM), or 64-QAM.

d) Forward error correction coding (convolutional coding) with a coding rate of 1/2, 2/3, or 3/4. Viterbi decoding will be implemented at the receiver.

e) 1 OFDM symbol per 4 μs (250, 000 sym/s)

The IEEE 802.11a PHY layer consists of two core sub-layers: Physical Layer Convergence Procedure (PLCP) and Physical Medium Dependent (PMD) layer. The PLCP maps the MAC frames onto the medium and serves as the boundary between the

MAC and PHY layers. The PMD layer carries out the actual transmission of these frames.

During transmission, multiple PHY sublayer Service Data Units (PSDUs) cascaded down from the MAC layer shall be appended with a PLCP preamble and header to form the PLCP Protocol Data Unit (PPDU). At the receiver, the PLCP preamble and header are retrieved and important information is extracted to help in the delivery of the PSDUs. For this thesis, the aim is to provide a software procedure in which PSDUs are converted to and from PPDUs. The format for the PPDU including the PLCP preamble, PLCP header, PSDU, tail bits, and pad bits are shown in Figure 3.

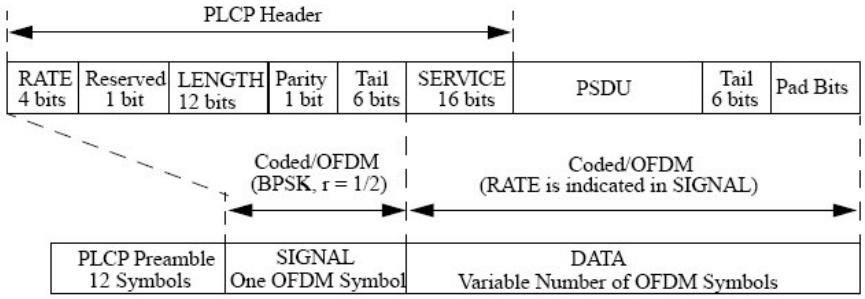

Figure 3. PPDU frame format (from: reference [3], Fig 107).

The LENGTH, RATE, reserved bit, parity bit and six "zero" tail bits are modulated to form a single OFDM symbol known as SIGNAL. This symbol is transmitted using BPSK modulation and a coding rate of $R = \frac{1}{2}$. The SERVICE field of the PLCP header, the PSDU, six "zero" tail bits and the necessary pad bits are modulated to form multiple OFDM symbols that is collectively known as DATA. DATA is transmitted at a data rate according to the RATE field and the LENGTH field determines the number of OFDM symbols in DATA. Hence, the RATE and LENGTH fields are critical in decoding the DATA part of the packet.

3. OSSIE Platform

The Open Source SCA Implementation::Embedded (OSSIE) is developed by the Mobile and Portable Radio Research Group (MPRG) at Virginia Tech as an open source SCA Core Framework solution. OSSIE was created to meet the need for a C++-based, open source SCA implementation that could be modified and adapted in a research

environment. The current version of OSSIE (0.5.0) is based on version 2.2.1 of the SCA specification. A detailed presentation of the OSSIE platform can be found in Jacob A. DePriest's thesis entitled "A Practical Approach to Rapid Prototyping of SCA Waveforms" at Virginia Tech [5]. From his thesis, the reader would be able to appreciate the OSSIE Waveform Developer (OWD) environment, specifically the following knowledge:

a) able to customize and design OSSIE **components** according to specific port implementation and inter-components threading strategies

b) able to set up **device** assignment to each component being developed

c) able to design a **waveform** using OWD to map a Radio design to the OSSIE software components available

This thesis is written with the assumption that the reader has certain prior knowledge about the C++ programming language, including object-oriented design. There are four important C++ files generated for each new component: <*Component Name*>*.h*, <*Component Name*>*.cpp*, *port_impl.h* and *port_impl.cpp*. These are where the functionalities are defined for the component. The content of these generated C++ files are modified to provide the actual functionality of a radio component.

a) *port_impl.h* and *port_impl.cpp*: implement the port communication between components, determining what is to be received and sent

b) <*Component Name*>*.h* and <*Component Name*>*.cpp*: 'brain' of the component, where its functionalities are programmed. Most of the post-generation codes are resided in the *process_data()* function call within <*Component Name*>*.cpp*.

B. CONCEPTUAL DESIGN

Following the principle of iterative and incremental development, five models have been developed, with each being more complex and built on the experiences gathered from the previous. The first three are exploratory models using MATLAB, which are relatively easy to build since many of the radio functionalities are already available as function calls. The fourth model builds on the success of the MATLAB

design. It emulates a Transmitter-Receiver (Tx-Rx) design using OSSIE but following closely the previous MATLAB model. The final model is the full scale OSSIE implementation of IEEE 802.11a PHY layer, which is the primary objective of this thesis work. A summary of the models is provided in Figure 4. The first four models are described here to demonstrate the incremental approach, while Section C presents the final full scale model.

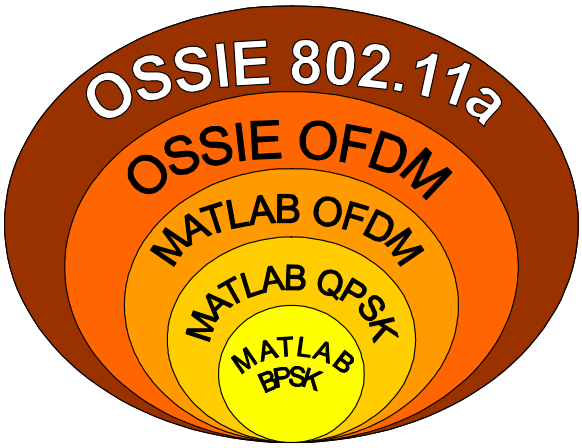

Figure 4. Incremental conceptual design.

1. MATLAB OFDM Models

An OFDM transmission design was implemented using MATLAB according to the source code recommended by Hiroshi and Ramjee [6]. There are three MATLAB models implemented, namely BPSK, QPSK and OFDM transceivers.

a. BPSK Modulation / Demodulation Transceiver

A block diagram of the design is shown in Figure 5. A simple BPSK modulation / demodulation was implemented to demonstrate the sequential flows of data between transmitter and receiver. A description of each component is provided in Table 1. The main MATLAB file that calls the various functions is *mainBPSK.m*.

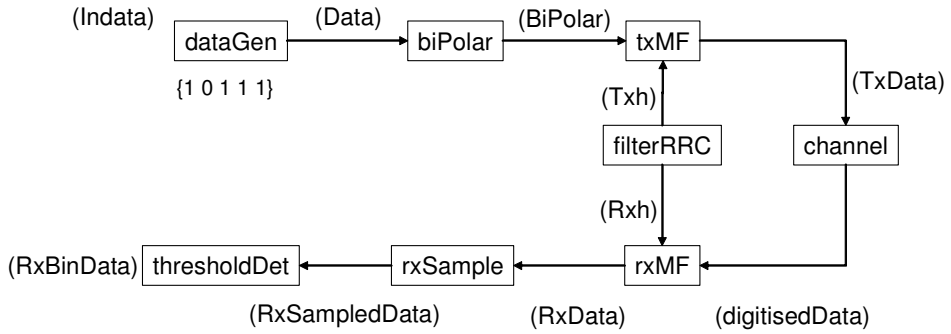

Figure 5. MATLAB BPSK transceiver model.

Filename	Purpose
dataGen	generate the tx binary data
biPolar	convert binary to polar data
txMF	Tx pulse-shaping using root raised cosine
filterRRC	generate coefficients of Nyquist filter
channel	model channel distortion (e.g. AWGN, fading)
rxMF	Rx pulse-shaping using root raised cosine
rxSample	sample the matched filter outputs
thresholdDet	threshold detector using Comparator

Table 1. MATLAB BPSK transceiver components description.

b. QPSK Modulation / Demodulation Transceiver

Next, a QPSK modulation / demodulation was implemented. The block diagram and components description are shown in Figure 6 and Table 2, respectively. The main MATLAB file that calls the various functions is *mainQPSK.m.*

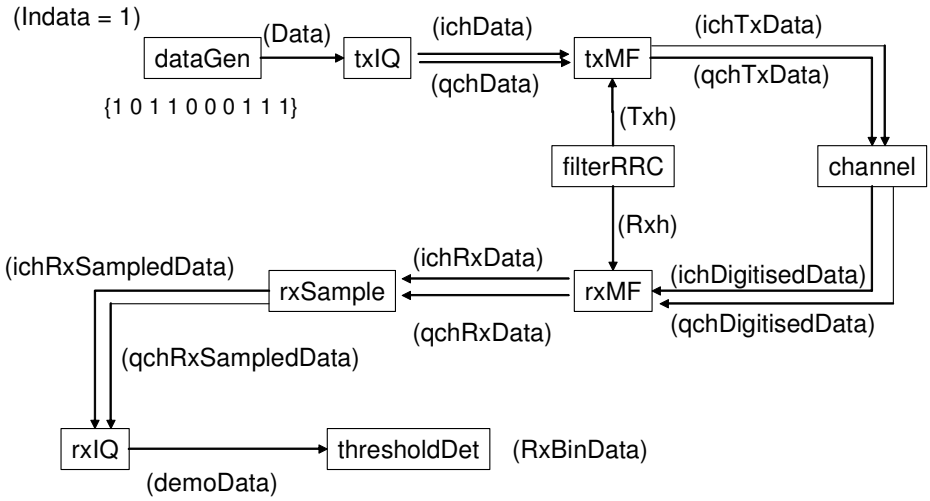

Figure 6. MATLAB QPSK transceiver model.

Filename	Purpose
dataGen	generate the tx binary data
txIQ	generate I and Q channel signals for QPSK (serial to parallel)
txMF	Tx pulse-shaping using root raised cosine
filterRRC	generate coefficients of Nyquist filter
channel	model channel distortion (e.g. AWGN, fading)
rxMF	Rx pulse-shaping using root raised cosine
rxSample	sample the matched filter outputs
rxIQ	demodulate I and Q channels signals for QPSK (parallel to serial)
thresholdDet	threshold detector using Comparator

Table 2. MATLAB QPSK transceiver components description.

c. OFDM Transceiver

The final design was an attempt to implement QPSK modulation with OFDM using MATLAB. The block diagram and component descriptions are shown in Figure 7 and Table 3, respectively. This design is the closest to the IEEE 802.11a PHY layer with many familiar functions that would eventually be 'converted' to components in OSSIE. These important functions include modulation mapping, normalization, inverse fast Fourier transform (IFFT) for OFDM, guard insert (or cyclic prefix insert) for the transmitter and demodulation mapping, unnormalization, fast Fourier transform (FFT) for OFDM and guard removal (or cyclic prefix removal) for the receiver. The main MATLAB file that calls the various functions is *mainOFDM.m.*

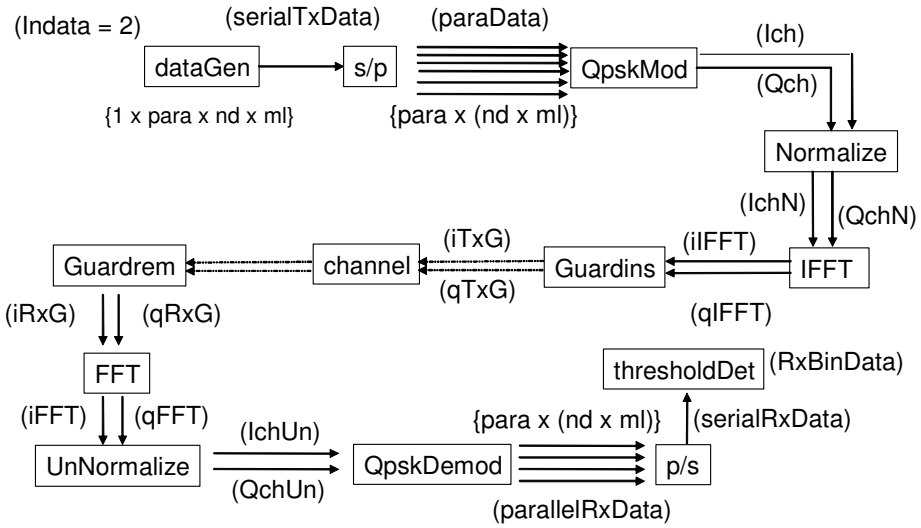

Figure 7. MATLAB OFDM transceiver model.

Filename	Purpose
OFDMdataGen	generate initial serial binary data
SerToPara	serial to parallel conversion
QpskMod	perform QPSK modulation
Normalize	Normalize the tx data
IFFT	IFFT for the Tx data
GiINS	insert guard interval into transmission signal
GiRem	remove guard interval from received signal
FFT	FFT for the Rx data
UnNormalize	UnNormalize the rx data
QpskDemod	perform QPSK demodulation
ParaToSer	parallel to serial conversion
ThresDet0	threshold detector using Comparator

Table 3. MATLAB OFDM transceiver components description.

The above three models provide a good stepping stone to the implementation of an OSSIE transceiver since they remove many of the complex software coding that is needed to implement the various functions. For example, IFFT and FFT are built-in functions in MATLAB, while direct coding is needed in C++ when OSSIE is used. More time can be spent on appreciating the data flow between components rather then worrying about coding the functionalities, i.e., understanding the functionalities is priority over coding the functionalities. All the necessary MATLAB files to implement the above three designs have been included in the reference CD.

2. OSSIE Tx-Rx OFDM Model

With a better understanding of the generic OFDM transceiver using MATLAB, the design shown in Figure 7 is ported over to OSSIE. All the functionalities are implemented as separate OSSIE components with the addition of two new components to demonstrate the inclusion of pilot subcarriers: carrier mapping (*crMapping*) and demapping (*crDemapping*). Their functionalities are summarized in Table 4. Detailed descriptions of each component shall be presented in the next two chapters under the full scale IEEE 802.11a implementation.

Filename	Purpose
crMapping	adds pilot subcarriers to the modulated data prior to IFFT processing
crDeMapping	removes pilot subcarriers from the FFT output prior to demodulation.

Table 4. OSSIE OFDM model additional components.

This intermediate model bridges the gaps between the MATLAB design and an OSSIE waveform where most of the functionalities have to be coded instead of depending on C++ built-in functions. Challenges in programming under the OSSIE environment begin to surface in this stage. Important programming experiences and lessons learned are described in the later chapters of this thesis. All the necessary OSSIE component and waveform files to implement the OFDM transceiver model have been included in the reference CD.

C. DETAILED DESIGN: OSSIE IEEE 802.11A TRANSCEIVER MODEL

The full scale 802.11a PHY layer is based on the IEEE standard 802.11a-1999 (Revision 2003) [3]. Design requirements are summarized in Section A.2 of this chapter. There are two core system architectures – transmitter and receiver. Both are implemented in software under the OWD environment.

1. Transmitter

The transmitter converts the binary inputs (especially the PSDU information from the MAC layer) into digitized PPDU frames to be passed through the DAC before up-conversion for RF transmission. According to Figure 3, the PPDU frame can be subdivided into three 'subframes', namely PLCP preamble (or just preamble), PLCP header excluding SERVICE (or just SIGNAL) and DATA. These represent the three separate 'modules' that shall be developed and appended to form the eventual transmitter PPDU frame. The components are developed either to carry out specific functions or to form the frames/subframes. The types of component needed are described in the components flow diagram of Figure 8.

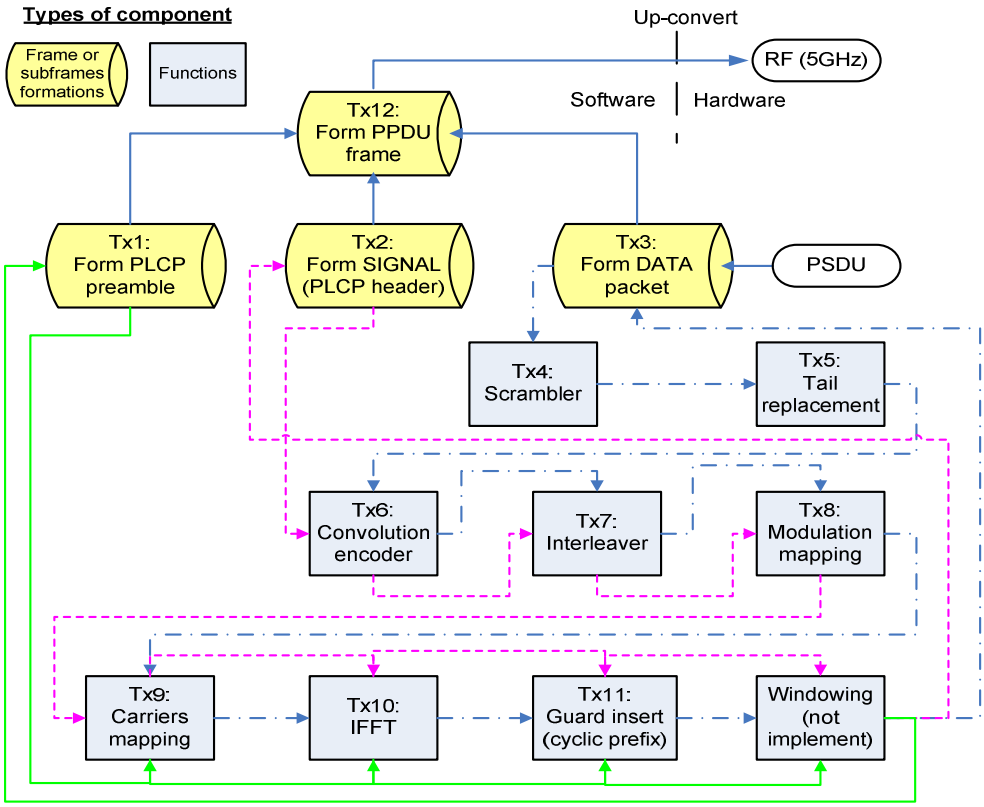

Figure 8. IEEE 802.11a Transmitter components flow diagram.

Figure 8 shows that twelve types of components (Tx1 – Tx12) are needed. Equally important is the fact that components of the same type are being reused in different subframes. For example, all three subframes employ the *carriers mapping* (Tx9), *IFFT* (Tx10) and *cyclic prefix* (Tx11) components, while only SIGNAL and Data subframes require the *convolution encoder* (Tx6), *interleaver* (Tx7) and *modulation mapping* (Tx8) components.

Note that *windowing* is not implemented as it is implementer specific and can be customized easily using software when needed. Time domain windowing was proposed in the IEEE 802.11a standard but it is just an informative rather than a mandatory approach. The implementer may choose other methods to achieve the purpose of smoothening the transitions between segments, such as frequency domain filtering.

Another way of representing the schematic is to describe the processing flow of each subframe separately as shown in Figure 9. This provides a better pictorial view of the functional flow for the formation of each subframe. It shows the sequential development of the PPDU frame. Figure 9 shows the quantity of each type of component that is needed to implement the transmitter. For example, four *carriers mapping* (Tx1.1.9, Tx1.2.9. Tx2.9 and Tx3.9) components are needed, while two *convolution encoder* (Tx2.6 and Tx3.6) components are necessary.

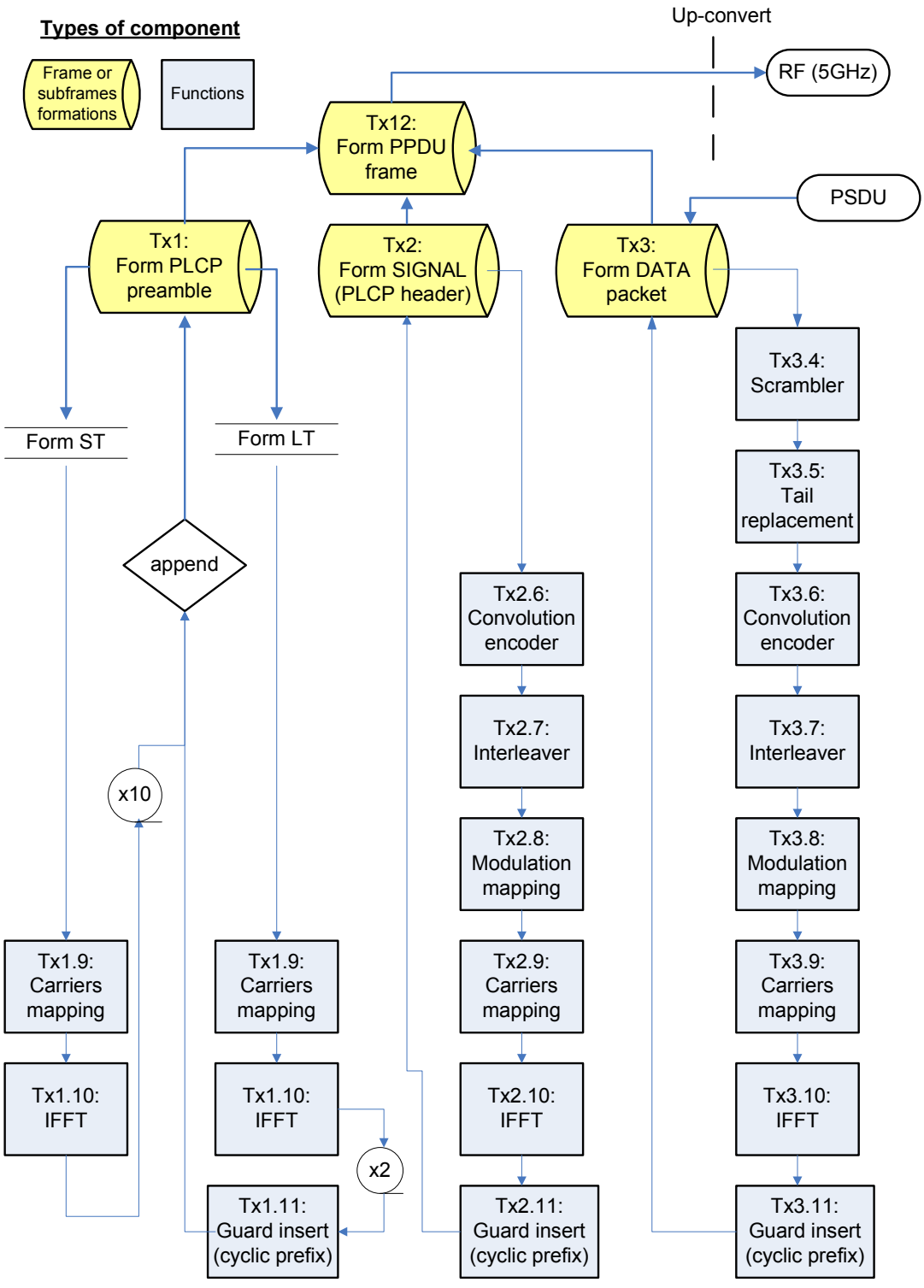

Figure 9. IEEE 802.11a Transmitter subframes flow diagram.

A summary of the functionalities of each component is provided in Table 5. The index abides by the following naming convention:

a) The first digit from the left represents the frame or subframe that the component belongs to. An exception is the preamble training sequences whereby Tx1.1.x represents component that forms the short training sequence and Tx1.2.x represents component that forms the long training sequence.

First digit	Frame / subframe
0	PSDU
1	preamble subframe
2	SIGNAL subframe
3	DATA subframe
12	PPDU

b) The last digit from the left represents the function of the component.

Last digit	Function
0	PSDU formation
1	preamble subframe formation
2	SIGNAL subframe formation
3	DATA subframe formation
4	Scrambler
5	Tail-replacement
6	Convolution encoder
7	Interleaver
8	Modulation
9	Carriers mapper
10	IFFT
11	Cyclic prefix
12	PPDU formation

For illustrations, Tx2.7 indicates a component under the SIGNAL subframe (2) that carries out the interleaving (7) function.

Index	Component	Functions
Preamble subframe		
Tx1	**preamble_map**	- initiate the Tx routine - form short training (ST) and long training (LT) sequence - send preamble (ST + LT) to PPDU
Tx1.1	**short training (ST)**	
Tx1.1.9	**ST_carrier_map**	- ST carrier mapping
Tx1.1.10	**ST_IFFT**	- ST IFFT
Tx1.2	**long training (LT)**	
Tx1.2.9	**LT_carrier_map**	- LT carrier mapping
Tx1.2.10	**LT_IFFT**	- LT IFFT
Tx1.2.11	**LT_cyclicPrefix**	- LT cyclic prefix append
SIGNAL subframe		
Tx2	**header_map (SIGNAL_map)**	- form SIGNAL (SIG) samples - send SIG to PPDU
Tx2.6	**SIG_conv_enc**	- SIG convolution encoding
Tx2.7	**SIG_interleaver**	- SIG interleaving
Tx2.8	**SIG_BPSK_mod**	- SIG BPSK modulation
Tx2.9	**SIG_carriers_map**	- SIG carriers mapping
Tx2.10	**SIG_IFFT**	- SIG IFFT
Tx2.11	**SIG_cyclicprefix**	- SIG cyclic prefix
DATA subframe		
Tx3	**data_map**	- form time data samples from PSDU - send DATA samples to PPDU
Tx3.4	**data_scrambler**	- scrambler the raw data
Tx3.5	**data_tail_replacement**	- replace tail with zeroes
Tx3.6	**data_conv_enc**	- data convolution encoding - data puncturing
Tx3.7	**data_interleaver**	- data interleaving
Tx3.8	**data_mod_map**	- data modulation mapping
Tx3.9	**data_carriers_map**	- data carriers mapping
Tx3.10	**data_IFFT**	- data IFFT
Tx3.11	**data_cyclicprefix**	- data cyclic prefix
Tx12	**PPDU_map**	- form PPDU frame from Preamble, SIG and DATA subframes for transmission
Tx0	**data_PSDU**	- input PSDU data

Table 5. IEEE 802.11a Transmitter components functionalities.

2. Receiver

The receiver carries out almost the inverse functions of the transmitter. In the receiver, digitized PPDU frames (passed down from the ADC after down-conversion from the RF front end) shall be converted into binary outputs from which the original PSDU information can be extracted. Like the transmitter, the receiver is comprised of three separate 'modules', namely preamble, SIGNAL and DATA subframes. The type of components needed are described in the components flow diagram of Figure 10. In comparison, fewer components are needed to implement the receiver than transmitter, but the receiver entails more complexity in the C++ algorithm.

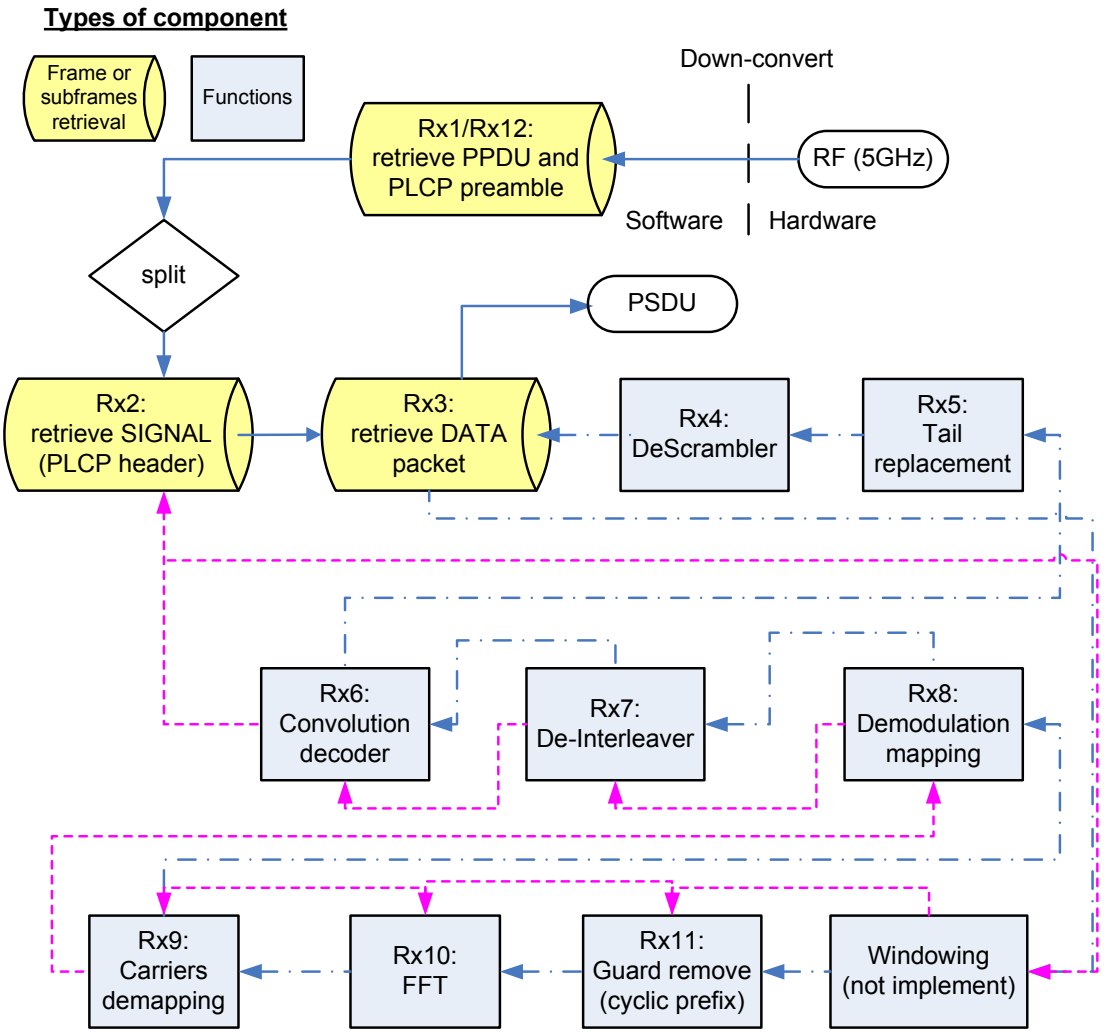

Figure 10. IEEE 802.11a Receiver components flow diagram.

Figure 10 shows that twelve types of components (Rx1 – Rx12) are needed. Note that Rx1 (PLCP preamble retrieval) and Rx12 (PPDU retrieval) are implemented in the same component. Synchronization is possible by assuming that the digitized samples received from the RF front end are compared to a fixed reference copy of the PLCP preamble sequence. Hence, from the PPDU stream, the entire received preamble sequence is identifiable, and this shall lead to the retrieval of the SIGNAL and DATA subframes. Detailed implementation of this component is described in Chapter IV, Section A2.

Like the transmitter, components of the same type are being reused in different subframes. For example, both SIGNAL and DATA subframes employ the *carriers demapping* (Rx9), *FFT* (Rx10) and *cyclic prefix removal* (Rx11) components, while only Data subframes require the *descrambler* (Rx4) component. Note that *windowing* is again not implemented, as it is implementer specific.

An alternate view of the schematic is to describe the processing flow of each subframe separately as shown in Figure 11. This shows the functional flow for the separation of each subframe and the eventual retrieval of the PSDU. Figure 11 shows the quantity of each type of component that is needed to implement the receiver. For example, two *carriers demapping* (Rx2.9 and Rx3.9) components are needed, while one *descrambler* (Rx3.4) component is needed.

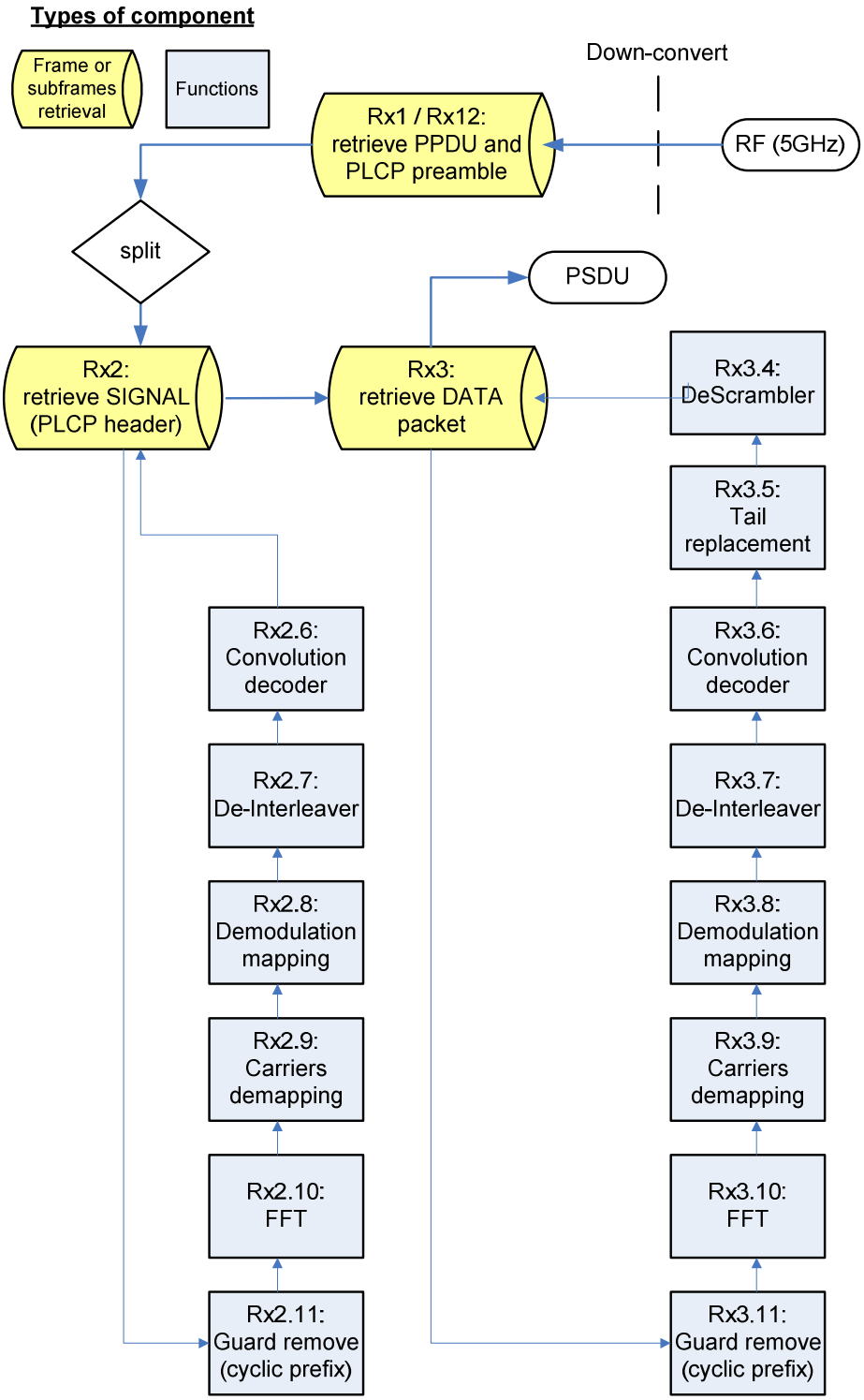

Figure 11. IEEE 802.11a Receiver subframes flow diagram.

The functionalities of each component are provided in Table 6. Similar to the transmitter, the receiver's component index abides by the following naming convention:

a) The first digit from the left represents the frame or subframe that the component belongs to.

First digit	Frame / subframe
0	Received digitized samples
1	preamble subframe
2	SIGNAL subframe
3	DATA subframe
12	PPDU

b) The last digit from the left represents the function of the component.

Last digit	Function
0	Digitized samples retrieval
1	preamble subframe retrieval
2	SIGNAL subframe retrieval
3	DATA subframe retrieval
4	Descrambler
5	Tail-replacement
6	Convolution decoder
7	Deinterleaver
8	Demodulation
9	Carriers demapper
10	FFT
11	Cyclic prefix removal
12	PPDU retrieval

For example, Rx3.7 indicates a component under the DATA subframe (3) that carries out the deinterleaving (7) function.

Index	Component	Functions
Preamble subframe		
Rx0	**Rx_data**	- received digitized data stream
Rx1 / Rx12	**PPDU_rx**	- extract the required digitized PPDU stream - removed preamble from PPDU - send stream for header removal
SIGNAL subframe		
Rx2	**Header_rx (SIGNAL_rx)**	- removed header from PPDU - send header for processing - extract RATE & LENGTH from SIG - send received data for processing
Rx2.11	**SIG_cyclicprefix_rem**	- SIG cyclic prefix removal
Rx2.10	**SIG_FFT**	- SIG FFT
Rx2.9	**SIG_carriers_demap**	- SIG carriers demapping
Rx2.8	**SIG_BPSK_demod**	- SIG BPSK demodulation
Rx2.7	**SIG_deinterleaver**	- SIG deinterleaving
Rx2.6	**SIG_conv_dec**	- SIG convolution decoding
DATA subframe		
Rx3	**data_rx**	- receive and send raw data for processing - receive and send PSDU data to MAC layer
Rx3.11	**data_cyclicprefix_rem**	- data cyclic prefix removal
Rx3.10	**data_FFT**	- data FFT
Rx3.9	**data_carriers_demap**	- data carriers demapping
Rx3.8	**data_demod_map**	- data demodulation mapping
Rx3.7	**data_deinterleaver**	- data deinterleaving
Rx3.6	**data_conv_dec**	- data dummy insertion - data convolution decoding
Rx3.5	**data_tail_replace**	- not required, encompass in descrambler
Rx3.4	**data_descrambler**	- descrambler the raw data

Table 6. IEEE 802.11a Receiver components functionalities.

In the next two chapters, the developmental details of each transmitter and receiver component are presented. The focus is on the C++ algorithm that implement the various functionalities. These functionalities are developed according to Table 5 and Table 6.

III. DEVELOP: TRANSMITTER

This and the next chapter provide the developmental details of the components in the transmitter and receiver to model the IEEE 802.11a PHY layer. In this chapter, the first three sections describe the three subframes of the transmitter: preamble, SIGNAL and DATA. The last section describes how the subframes are concatenated to form the PPDU. Components are described according to the inter-linkages of the input-output (I/O) ports and the functional implementation in C++ code.

The transmitter converts the binary inputs (especially the PSDU information from the MAC layer) into digitized PPDU frames to be sent through the DAC before up-conversion for RF transmission. The incremental development is discussed here starting with the preamble subframe, followed by SIGNAL subframe and, finally, the overall PPDU frame with the inclusion of the DATA subframe.

A. PREAMBLE

The PLCP preamble subframe consists of ten repetitions of a short training (ST) sequence and two repetitions of a long training (LT) sequence, preceded by a guard interval (cyclic prefix). The format for the PLCP preamble subframe is presented in Figure 12. Table 7 summarizes the OSSIE components needed to form the preamble subframe.

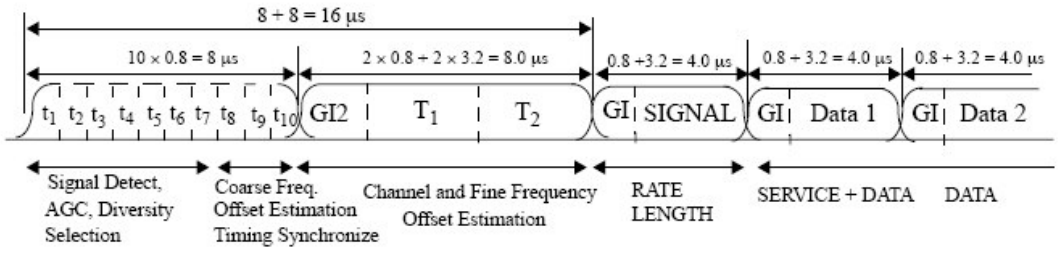

Figure 12. PPDU frame structure and timing. (from: reference [3], Fig. 110).

Index	Component	Functions
Tx1	**preamble_map**	- initiate the Tx routine - form short training (ST) and long training (LT) sequence - send preamble (ST + LT) to form PPDU
Tx1.1	**short training (ST)**	
Tx1.1.9	**ST_carriers_map**	- ST carriers mapping
Tx1.1.10	**ST_IFFT**	- ST IFFT
Tx1.2	**long training (LT)**	
Tx1.2.9	**LT_carriers_map**	- LT carriers mapping
Tx1.2.10	**LT_IFFT**	- LT IFFT
Tx1.2.11	**LT_cyclicPrefix**	- LT cyclic prefix append

Table 7. IEEE 802.11a Transmitter preamble subframe components functionalities.

1. Tx1: Preamble Mapping (Assembly Controller)

Component name: ***preamble_map***

Port design: *preamble_map* is the assembly controller for the transmitter. Assembly controller is the only component within each model where the *start()* function is being called when the waveform is first loaded. When it is time to start the radio, the assembly controller's *start()* function shall initiate the transmitter software routine to form the PPDU frame. It has a total of 2 input ports where data is flowing into the component (*ST_input* and *LT_input*) and 3 output ports where data is flowing out of the component (*ST_processing*, *LT_processing* and *preamble subframe*). Figure 13 shows the I/O distribution of the component.

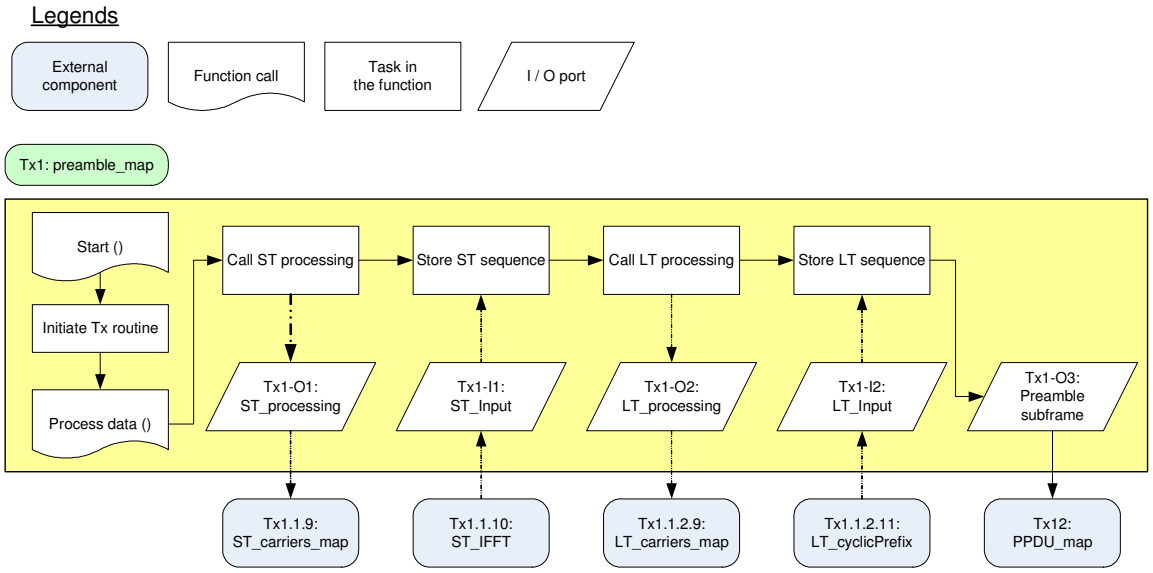

Figure 13. *preamble_map* port and functional flow.

Functional design: *preamble_map* carries out three main functions: (1) initiate the transmitter routine, (2) form ST and LT sequences, and (3) append the two sequences and form the preamble subframe. This sequential functional flow is shown in Figure 13. After the initiation from the *start()* function, *process_data()* is called to start the ST processing. The *ST_processing* output port is activated to push the relevant packets to another component, *ST_carriers_map*, which carries on with the formation of ST sequences. Eventually, the processed ST sequence shall be routed back to *preamble_map* component from *ST_IFFT* component via the *ST_input* input port. The same approach is carried out to generate the LT sequence by using the *LT_processing* output port and *LT_input* input port to connect to *LT_carriers_map* and *LT_cyclicPrefix* component respectively. With the two sequences generated and attached to each other, the final data is pushed to *PPDU_map* component via the *preamble_subframe* output port.

2. Tx1.1.9: Carriers Mapping (ST)

Component name: ***ST_carriers_map***

Port design: *ST_carriers_map* has 1 input port and 1 output port. Figure 14 shows the I/O distribution of the component.

Functional design: Its main function is to re-index and normalize the 52 subcarriers of the initial short training sequence as part of the 64 frequency samples (by introducing guard bands as stipulated in the IEEE standard) to serve as input into the IFFT component. The functional flow is shown in Figure 14.

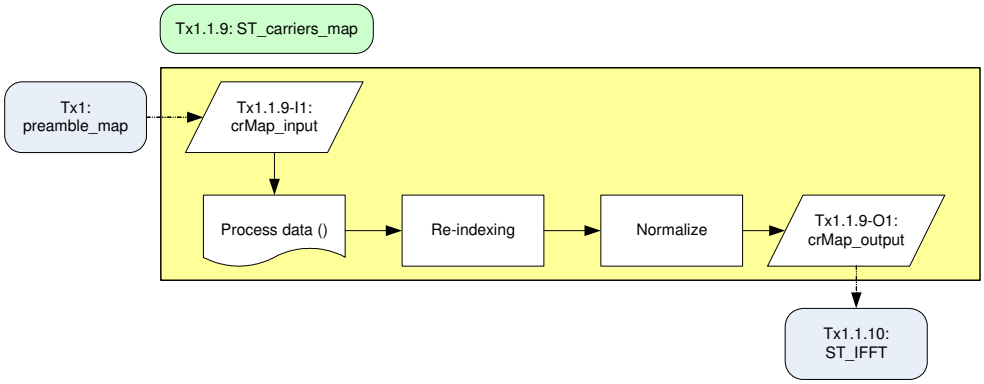

Figure 14. *ST_carriers_map* port and functional flow.

3. Tx1.1.10: IFFT (ST)

Component name: ***ST_IFFT***

Port design: *ST_IFFT* has 1 input port and 1 output port. Figure 15 shows the I/O distribution of the component.

Functional design: The component emulates OFDM processing through a software IFFT algorithm. From *ST_carriers_map*, 64 frequency samples shall be sent through *ST_IFFT* to convert to 64 time samples. In this implementation the Decimation-In-Time (DIT) Permutated Input - Natural Output (PINO) IFFT algorithm [7] is selected as it is reasonably easy to comprehend and code. The time samples need to be duplicated 10 times to form the necessary sequence length for the preamble subframe. The functional flow is shown in Figure 15. The DIT PINO IFFT algorithm has been singled out as special interest component that shall be described in Chapter V.

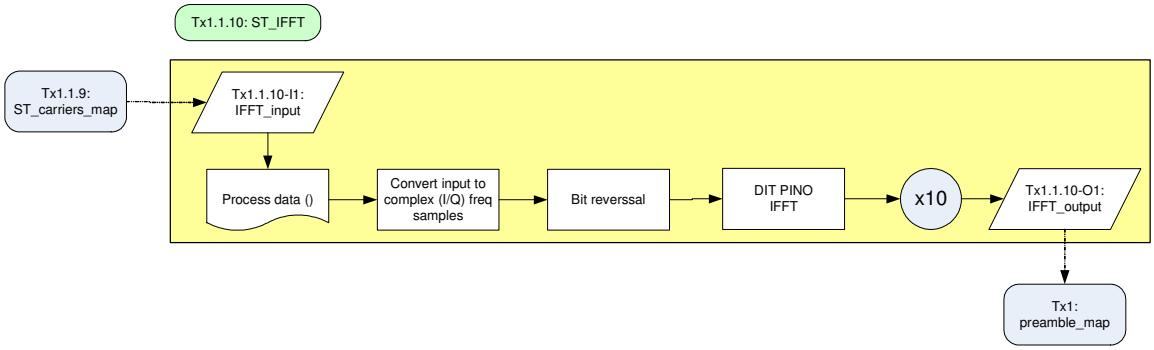

Figure 15. *ST_IFFT* port and functional flow.

4. Tx1.2.9: Carriers Mapping (LT)

Component name: ***LT_carriers_map***

Port design: *LT_carriers_map* has 1 input port and 1 output port. Figure 16 shows the I/O distribution of the component.

Functional design: Similar to its ST counterpart, *LT_carriers_map* re-indexes the 52 subcarriers of the initial long training sequence as part of the 64 frequency samples (by introducing guard bands as stipulated in the IEEE standard) to serve as input into the IFFT component. The functional flow is shown in Figure 16.

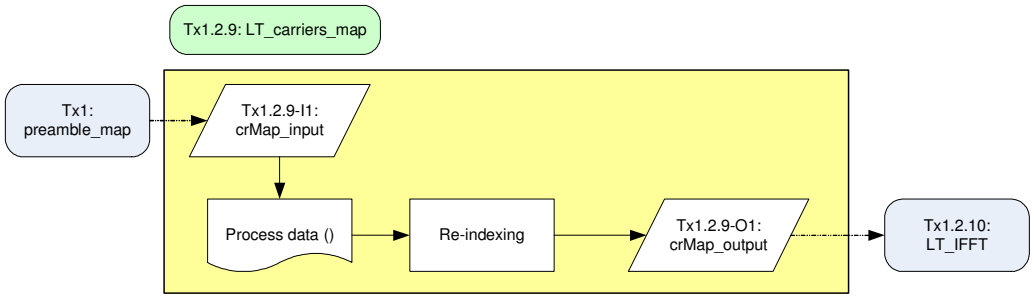

Figure 16. *LT_carriers_map* port and functional flow.

5. Tx1.2.10: IFFT (LT)

Component name: ***LT_IFFT***

Port design: *LT_IFFT* has 1 input port and 1 output port. Figure 17 shows the I/O distribution of the component.

Functional design: Like *ST_IFFT*, this component emulates OFDM processing through software IFFT algorithm. From *LT_carriers_map*, 64 frequency samples will be sent through *LT_IFFT* to convert to 64 time samples. The DIT PINO IFFT algorithm shall be described in Chapter V. The time samples are duplicated twice before inserting a cyclic prefix in the next component. The functional flow is shown in Figure 17.

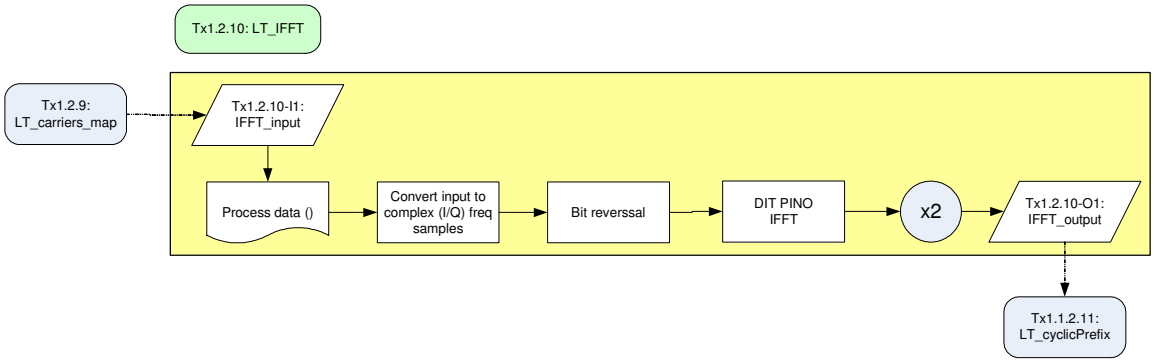

Figure 17. *LT_IFFT* port and functional flow.

6. Tx1.2.11: Cyclic Prefix (LT)

Component name: ***LT_cyclicPrefix***

Port design: *LT_cyclicPrefix* has 1 input port and 1 output port. Figure 18 shows the I/O distribution of the component.

Functional design: *LT_cyclicPrefix* prefixes half the length of one full IFFT time samples to the data to form the LT sequence. This sequence is forwarded to *preamble_map* and appended to the ST sequence as preamble subframe. The functional flow is shown in Figure 18.

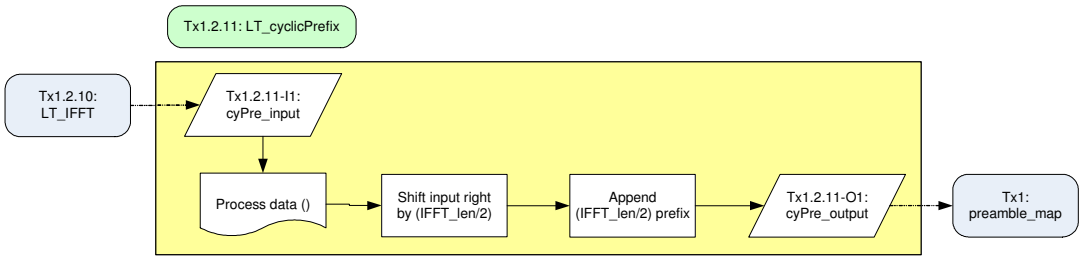

Figure 18. *LT_ cyclicPrefix* port and functional flow.

B. SIGNAL

The SIGNAL symbol consists of RATE and LENGTH fields that are encoded by a convolutional code of $R = \frac{1}{2}$. It is subsequently mapped onto a single OFDM symbol with BPSK sub-channel modulation. The encoding of the SIGNAL field into an OFDM symbol follows the procedures: convolutional encoding, interleaving, BPSK modulation, pilot insertion (guard band), IFFT and appending a cyclic prefix at a data rate of 6 Mbits/s. Unlike the DATA subframe, the information bits of the SIGNAL field are not scrambled. Table 8 summarizes the components needed to form the SIGNAL subframe.

Index	Component	Functions
Tx2	**header_map (SIGNAL_map)**	- form SIGNAL (SIG) samples - send SIG to form PPDU
Tx2.6	**SIG_conv_enc**	- SIG convolution encoding
Tx2.7	**SIG_interleaver**	- SIG interleaving
Tx2.8	**SIG_BPSK_mod**	- SIG BPSK modulation
Tx2.9	**SIG_carriers_map**	- SIG carriers mapping
Tx2.10	**SIG_IFFT**	- SIG IFFT
Tx2.11	**SIG_cyclicprefix**	- SIG cyclic prefix

Table 8. IEEE 802.11a Transmitter SIGNAL subframe components functionalities.

1. Tx2: SIGNAL Mapping

Component name: ***SIGNAL_map***

Port design: *SIGNAL_map* has a total of 2 input ports and 2 output ports. Figure 19 shows the I/O distribution of the component.

Functional design: *SIGNAL_map* carries out the function of concatenating important parameters (especially RATE and LENGTH) and sending it for processing to form an OFDM symbol at a data rate of 6 Mbits/s. At this data rate, the modulation type and data structure are fixed as shown in Table 9. RATE and LENGTH can either be passed down from the MAC layer or entered by other means.

The final symbol shall form the SIGNAL subframe to be transmitted as part of PPDU. The SIGNAL field is composed of 24 bits, as shown in Figure 20. Bits 0 to 3 shall represent the RATE. Bit 4 is reserved for future use. Bits 5 to 16 shall represent the LENGTH field, with the least significant bit (LSB) being transmitted first. The component functional flow is shown in Figure 19.

Data rate $(Mbits/s)$	Modulation	Coding rate (R)	Coded bits Per subcarrier (N_{BPSC})	Coded bits per OFDM symbol (N_{CBPS})	Data bits per OFDM symbol (N_{DBPS})
6	BPSK	1/2	1	48	24

Table 9. Rate-dependent parameters: 6 Mbits/s.

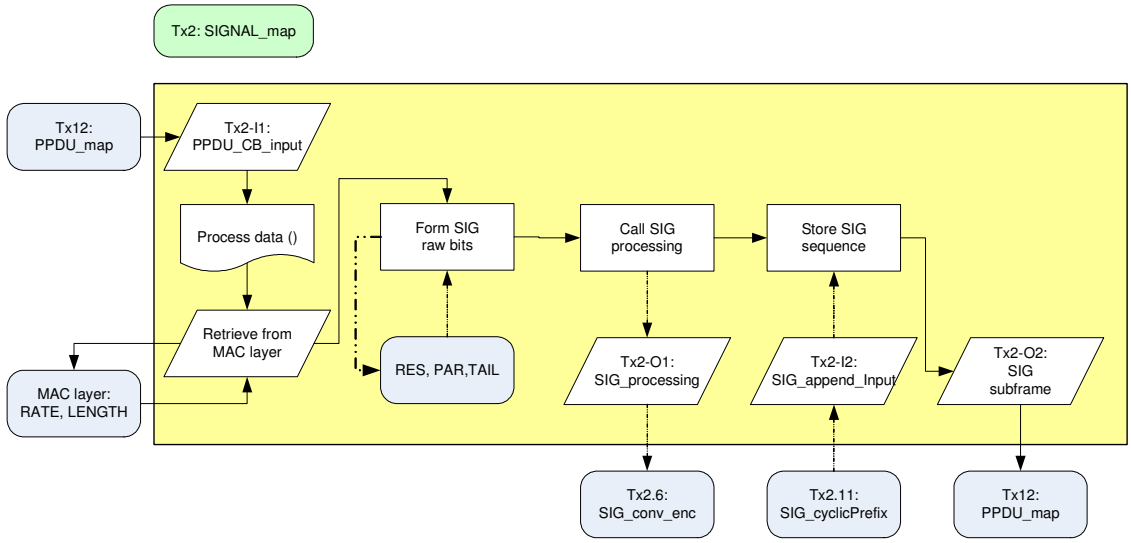

Figure 19. *SIGNAL_map* port and functional flow.

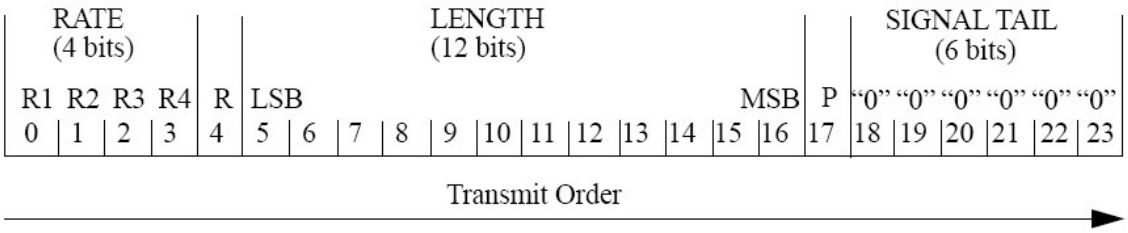

Figure 20. composition of SIGNAL field (from: reference [3], Fig. 111).

2. Tx2.6: Convolutional Encoder (SIG)

Component name: ***SIG_conv_enc***

Port design: *SIG_conv_enc* has 1 input port and 1 output port. Figure 22 shows the I/O distribution of the component.

Functional design: The SIGNAL field is coded with a convolutional encoder to a coding rate of $R = \frac{1}{2}$. The convolutional encoder is shown in Figure 21 with the two generator polynomials stated in the IEEE 802.11a standard ($g_A = 133_8$ and $g_B = 171_8$) and a fixed rate $R_{enc} = \frac{1}{2}$. The output data "A" is transmitted from the encoder before the output bit denoted as "B". The functional flow is shown in Figure 22. Since there are six memory elements (constraint length ν of 7) in the shift register, this explains the requirement of having six "zero" tail bits in the SIGNAL field prior to encoding.

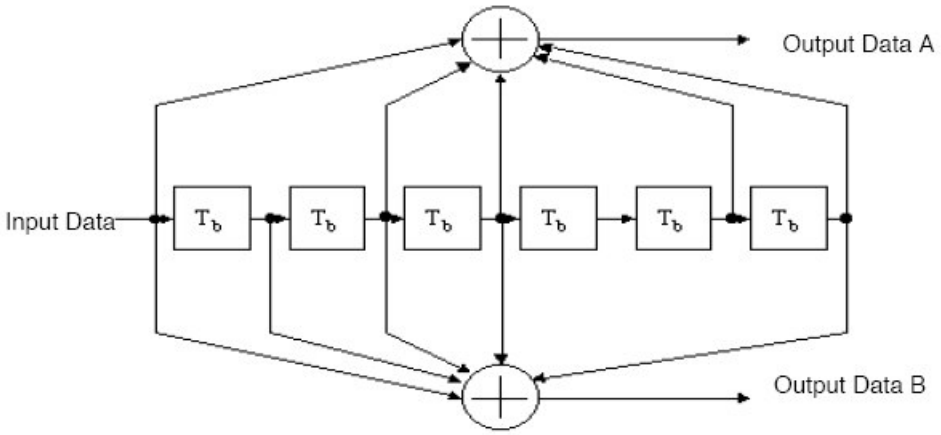

Figure 21. convolutional encoder ($\nu = 7$) (from: reference [3], Fig. 114).

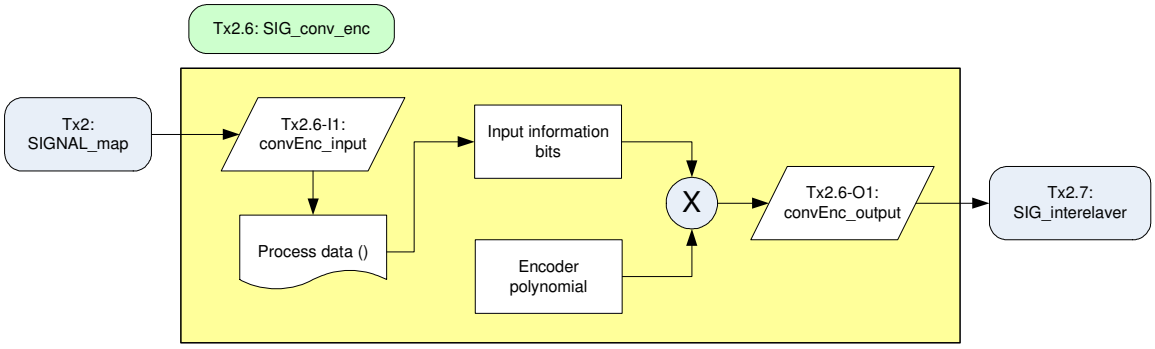

Figure 22. *SIG_conv_enc* port and functional flow.

3. Tx2.7: Interleaver (SIG)

Component name: *SIG_interleaver*

Port design: *SIG_interleaver* has 1 input port and 1 output port. Figure 23 shows the I/O distribution of the component.

Functional design: All data bits are passed through a block interleaver after the encoding process. The interleaver has a block size equals to the number of coded bits in a single OFDM symbol, N_{CBPS} (see Table 9). The interleaver consists of two different permutations. The first permutation ensures that adjacent coded bits do not map onto adjacent subcarriers (refers to Equation 1 for the mapping). The second permutation ensures that adjacent coded bits are alternate between less and more significant bits after the mapping to prevent long runs of low reliability bits [3] (refers to Equation 2 for the mapping). Note that k, i and j refer to the index of the coded bit before the first, before the second and after the second permutation, respectively. The function *floor(.)* denotes the largest integer not exceeding the parameter. The value of s is derived from the number of coded bits per subcarrier, N_{BPSC}, according to $s = \max\left(\frac{N_{BPSC}}{2}, 1\right)$. Hence, $s = 1$. The functional flow is shown in Figure 23.

$$i = \left(\frac{N_{CBPS}}{16}\right)(k \bmod 16) + floor\left(\frac{k}{16}\right) \qquad i,k = 0,1,\ldots,N_{CBPS}-1 \tag{1}$$

$$j = s \times floor\left(\frac{i}{s}\right) + \left(i + N_{CBPS} - floor\left(16 \times \frac{i}{N_{CBPS}}\right)\right) \bmod s \quad i,j = 0,1,\ldots,N_{CBPS}-1 \tag{2}$$

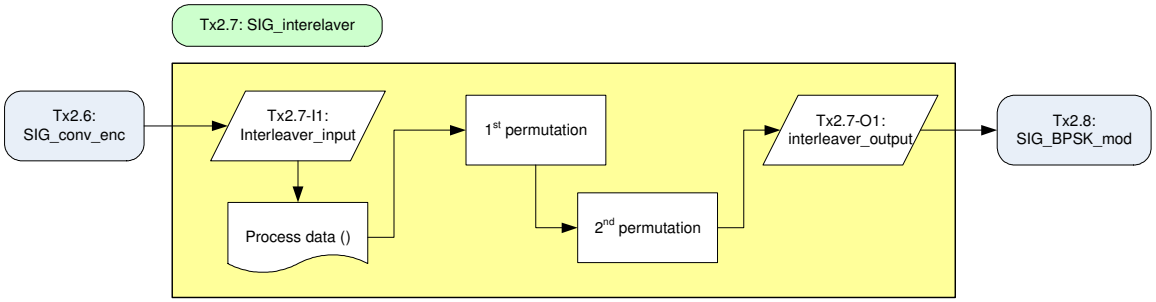

Figure 23. *SIG_interleaver* port and functional flow.

4. Tx2.8: BPSK Modulation (SIG)

Component name: ***SIG_BPSK_mod***

Port design: *SIG_BPSK_mod* has 1 input port and 1 output port. Figure 29 shows the I/O distribution of the component.

Functional design: The SIGNAL OFDM subcarriers shall be modulated by using BPSK modulation. The encoded and interleaved binary input data shall be converted into complex BPSK constellation points. The output values for a modulator are formed by multiplying the resulting I and Q channel values by a normalization factor K_{MOD} to achieve the same average power for all mappings. For BPSK modulation, K_{MOD} is unity, and normalization is not necessary in this specific modulator.

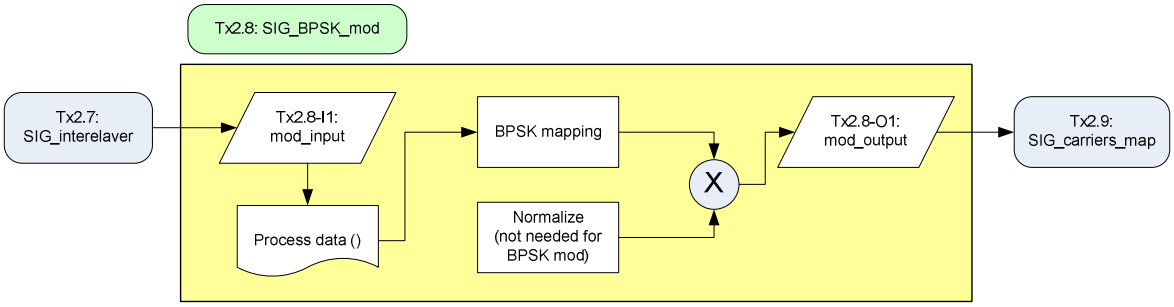

Figure 24. *SIG_BPSK_mod* port and functional flow.

5. Tx2.9: Carriers Mapping (SIG)

Component name: ***SIG_carriers_map***

Port design: *SIG_carriers_map* has 1 input port and 1 output port. Figure 25 shows the I/O distribution of the component.

Functional design: Four pilot tones are inserted to form 52 subcarriers. Similar to its preamble counterpart, *SIG_carriers_map* re-indexes this 52 subcarriers after BPSK modulation as part of the 64 frequency samples (by introducing guard bands as stipulated in the IEEE standard) to serve as input into the IFFT component. The functional flow is shown in Figure 25.

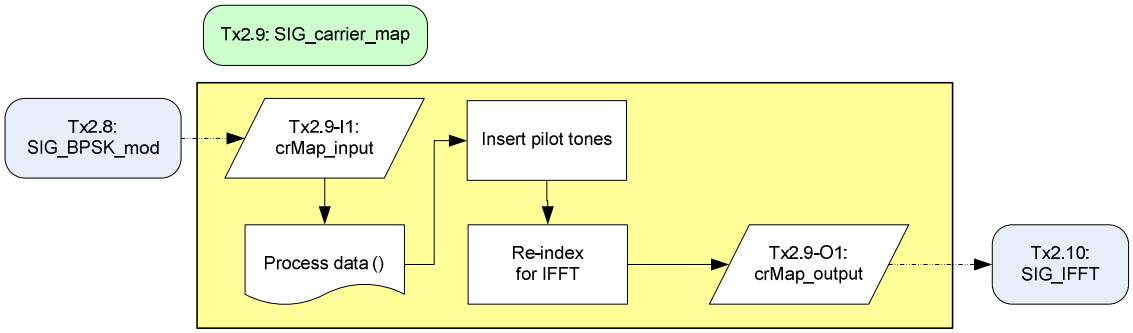

Figure 25. *SIG_carriers_map* port and functional flow.

6. Tx2.10: IFFT (SIG)

Component name: ***SIG_IFFT***

Port design: *SIG_IFFT* has 1 input port and 1 output port. Figure 26 shows the I/O distribution of the component.

Functional design: From *SIG_carriers_map*, 64 frequency samples will be sent through *SIG_IFFT* to convert to 64 time samples. The DIT PINO IFFT algorithm shall be described in Chapter V. The functional flow is shown in Figure 26.

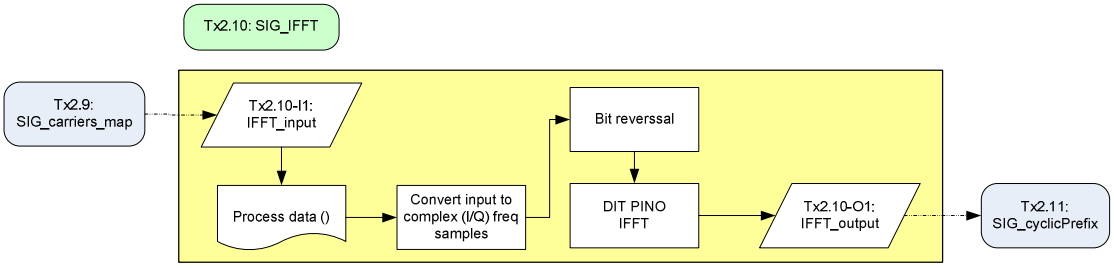

Figure 26. *SIG_IFFT* port and functional flow.

7. Tx2.11: Cyclic Prefix (SIG)

Component name: ***SIG_cyclicPrefix***

Port design: *SIG_cyclicPrefix* has 1 input port and 1 output port. Figure 27 shows the I/O distribution of the component.

Functional design: *SIG_cyclicPrefix* prefixes a quarter of the length of the IFFT time samples to the data to form the SIGNAL subframe. This sequence shall be forwarded to *SIGNAL_map* to be concatenated as part of PPDU. The functional flow is shown in Figure 27.

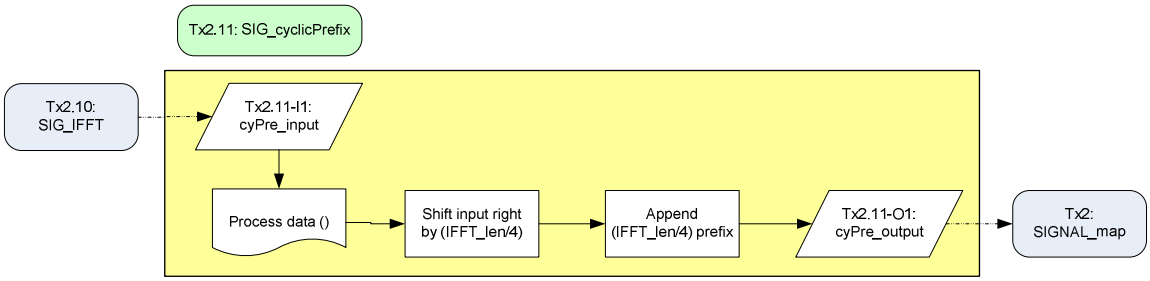

Figure 27. *SIG_cyclicPrefix* port and functional flow.

C. DATA

The DATA field contains the SERVICE field, the PSDU, the TAIL bits, and the PAD bits (when necessary). The SERVICE field has 16 bits as shown in Figure 28. The first 7 bits are set to zeros to synchronize the descrambler over at the receiver. The remaining 9 bits are reserved for future use and are also set to zero. The PSDU tail bit field shall be six bits of "0", which serve the function of returning the convolutional encoder to the "zero state" (similar function as the 6 tail bits of SIGNAL field). The PPDU tail bit field shall be maintained by replacing six scrambled "zero" bits following the end of the message (which may not be "zero") with six non-scrambled "zero" bits.

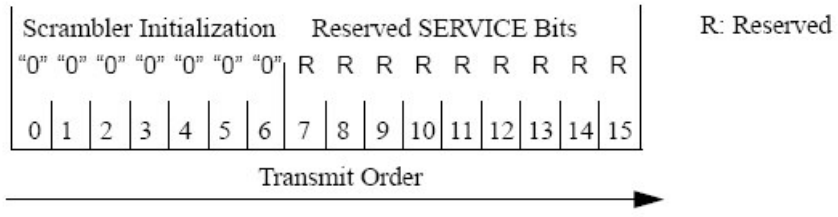

Figure 28. Composition of SERVICE field (from: reference [3], Fig. 112).

Besides all the components (functionalities) under the SIGNAL subframe, forming the DATA subframe requires an additional initial procedure of scrambling the information. Table 10 summarizes the components needed to form the DATA subframe.

Index	Component	Functions
Tx3	**data_map**	- form time data samples from PSDU - send DATA samples to form PPDU
Tx3.4	**data_scrambler**	- scramble the raw data
Tx3.5	**data_tail_replacement**	- replace tail with zero
Tx3.6	**data_conv_enc**	- data convolutional encoding - data puncturing
Tx3.7	**data_interleaver**	- data interleaving
Tx3.8	**data_mod_map**	- data modulation mapping
Tx3.9	**data_carriers_map**	- data carriers mapping
Tx3.10	**data_IFFT**	- data IFFT
Tx3.11	**data_cyclicprefix**	- data cyclic prefix
Tx0	**data_PSDU**	- input PSDU data

Table 10. IEEE 802.11a Transmitter DATA subframe components functionalities.

1. Tx3: DATA Mapping

Component name: ***DATA_map***

Port design: *DATA_map* has a total of 3 input ports and 3 output ports. Figure 29 shows the I/O distribution of the component.

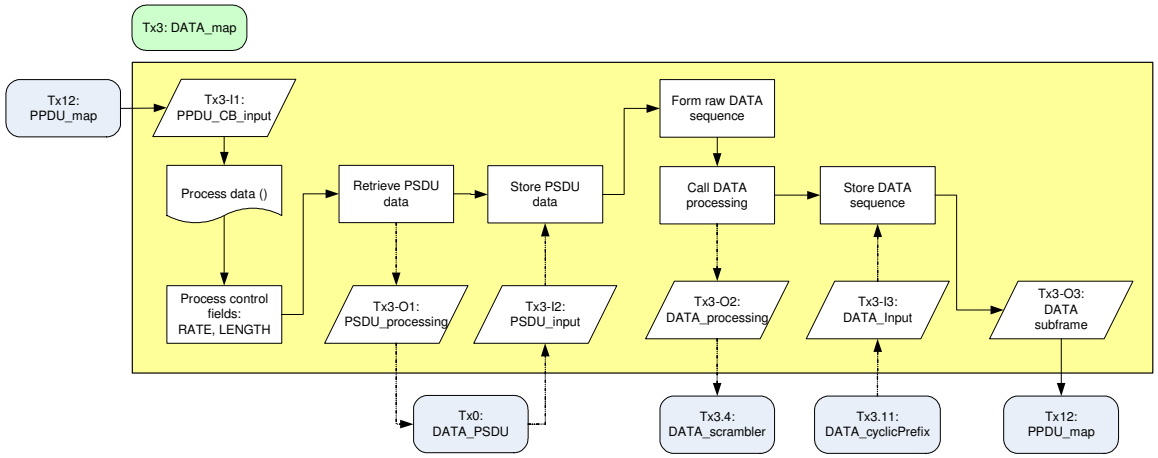

Figure 29. *DATA_map* port and functional flow.

Functional design: *DATA_map* is the heart of DATA processing at the PHY layer where PSDUs from the MAC layer are processed into time samples to be concatenated and formed the PPDU. The first function in this component is to extract important information from the two control fields: RATE and LENGTH. From RATE (data

transmission rate), the modulation type and coding parameters are defined as shown in Table 11.

Data rate $(Mbits/s)$	Modulation	Coding rate (R)	Coded bits Per subcarrier (N_{BPSC})	Coded bits per OFDM symbol (N_{CBPS})	Data bits per OFDM symbol (N_{DBPS})
6	BPSK	1/2	1	48	24
9	BPSK	3/4	1	48	36
12	QPSK	1/2	2	96	48
18	QPSK	3/4	2	96	72
24	16-QAM	1/2	4	192	96
36	16-QAM	3/4	4	192	144
48	64-QAM	2/3	6	288	192
54	64-QAM	3/4	6	288	216

Table 11. Rate-dependent parameters.

The LENGTH field determines the size of PSDU to be sent in a PPDU frame, which is user dependent. The length of the message is extended to be a multiple of N_{DBPS}, while ensuring the number of bits in the DATA field is a multiple of N_{CBPS}. This is possible by inserting PAD bits to the PPDU frame. The number of OFDM symbols, N_{SYM}, the number of bits in the DATA field, N_{DATA}, and the number of PAD bits, N_{PAD}, are computed according to Equation 3, 4 and 5 respectively.

$$N_{SYM} = ceiling\left((16 + 8 \times LENGTH + 6)/N_{DBPS}\right) \quad (3)$$

$$N_{DATA} = N_{SYM} \times N_{DBPS} \quad (4)$$

$$N_{PAD} = N_{DATA} - (16 + 8 \times LENGTH + 6) \quad (5)$$

The *ceiling(.)* function returns the smallest integer value greater than or equal to its argument value. The appended PAD bits are set to "zero" and shall be scrambled with the other bits in the DATA field.

From the LENGTH field, *DATA_map* carries out its second function of retrieving the PSDU from the MAC layer (simulated by *DATA_PSDU*). After that, the DATA bits are sent for processing into time samples. The OFDM symbols are transferred to *PPDU_map* and form the PPDU. The functional flow is shown in Figure 29.

2. Tx3.4: Scrambler (DATA)

Component name: ***DATA_scrambler***

Port design: *DATA_scrambler* has 1 input port and 1 output port. Figure 31 shows the I/O distribution of the component.

Functional design: The DATA subframe shall be scrambled with a length-127 scrambler stipulated in the IEEE 802.11a standard (as illustrated in Figure 30). The scrambler is set to a pseudo random non-zero initial state when first applied to the input data. This initial state must be the same as that for the receiver descrambler. The functional flow is shown in Figure 31. X^1 to X^7 represent the seven shift registers and the outputs from the fourth $\left(X^4\right)$ and seventh $\left(X^7\right)$ registers are mod-2 added and cycled back to the shift registers. This mod-2 output is also used to scramble the input data by mod-2 addition with the input data.

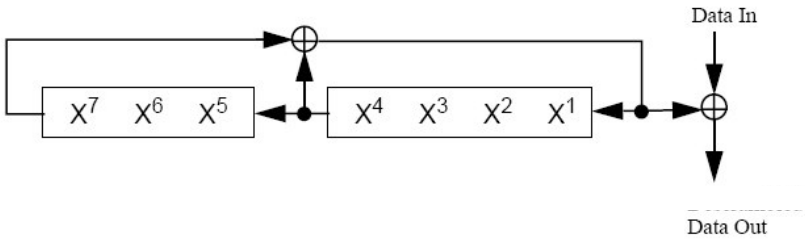

Figure 30. DATA scrambler (from: reference [3], Fig. 113).

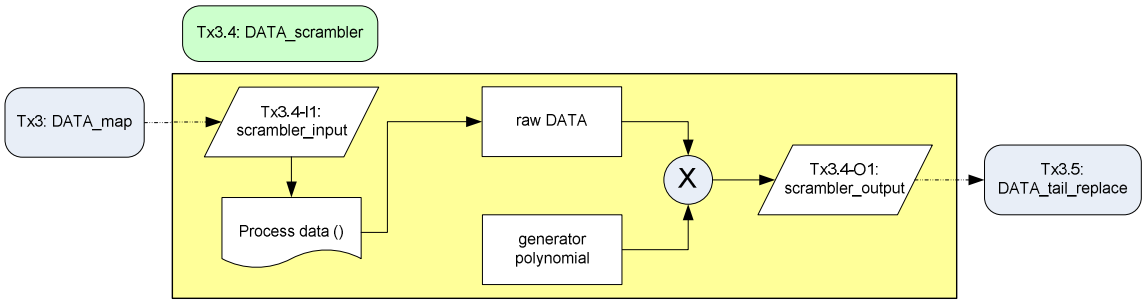

Figure 31. *DATA_ scrambler* port and functional flow.

3. Tx3.5: Tail Replacement (DATA)

Component name: ***DATA_tail_replacement***

Port design: *DATA_tail_replacement* has 1 input port and 1 output port. Figure 32 shows the I/O distribution of the component.

Functional design: The DATA tail bit field shall be reproduced by replacing six scrambled "zero" bits following the message end with six non-scrambled "zero" bits. The six "zero" tail bits are needed to return the convolution encoder (next component) to the state of all "zero". The functional flow is shown in Figure 32.

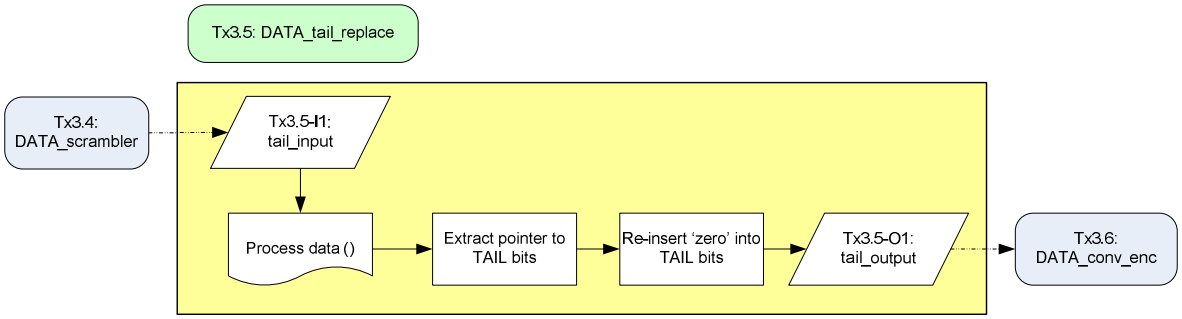

Figure 32. *DATA_ tail_replacement* port and functional flow.

4. Tx3.6: Convolutional Encoder (DATA)

Component name: ***DATA_conv_enc***

Port design: *DATA_conv_enc* has 1 input port and 1 output port. Figure 34 shows the I/O distribution of the component.

Functional design: The DATA subframe shall be coded with a convolutional encoder of coding rate $R = \frac{1}{2}$, $\frac{2}{3}$, or $\frac{3}{4}$, depending on the data rate being transmitted (see Table 11). The convolutional encoder is shown in Figure 21 with the two generator polynomials stated in the IEEE 802.11a standard ($g_A = 133_8$ and $g_B = 171_8$) and a fixed rate $R_{enc} = \frac{1}{2}$. The bit denoted as "A" is transmitted from the encoder before the bit denoted as "B". The process of puncturing is employed to obtain the higher coding rates. Puncturing is a procedure for removing some of the encoded bits in the transmitter after passing through the $R_{enc} = \frac{1}{2}$ generator polynomials. The puncturing patterns are illustrated in Figure 33 for $R = \frac{2}{3}$ and $\frac{3}{4}$. Note that no puncturing is needed for $R = \frac{1}{2}$. The functional flow of the component is shown in Figure 34.

Puncturing code: R=3/4

Source bits	x0	x1	x2	x3	x4	x5	x6	x7	x8		
Encoded bits	A0	A1	A2	A3	A4	A5	A6	A7	A8		
	B0	B1	B2	B3	B4	B5	B6	B7	B8		
Transmitted bits	A0	B0	A1	B2	A3	B3	A4	B5	A6	B6	A7	B8

Puncturing code: R=2/3

Source bits	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9					
Encoded bits	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9					
	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9					
Transmitted bits	A0	B0	A1	A2	B2	A3	A4	B4	A5	A6	B6	A7	A8	B8	A9

punctured bit

Figure 33. *DATA_conv_enc* puncturing patterns (after: reference [3], Fig. 115).

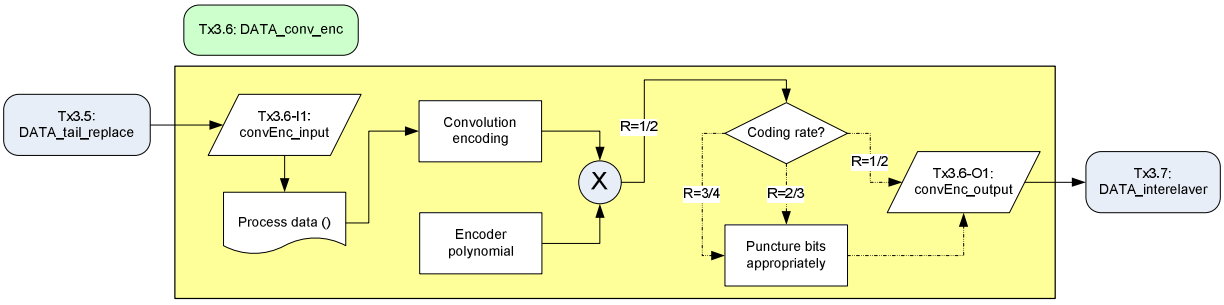

Figure 34. *DATA_conv_enc* port and functional flow.

5. Tx3.7: Interleaver (DATA)

Component name: ***DATA_interleaver***

Port design: *DATA_interleaver* has 1 input port and 1 output port. Figure 35 shows the I/O distribution of the component.

Functional design: All data bits are passed through a block interleaver after the encoding process. The interleaver has a block size equals to the number of coded bits in a single OFDM symbol, N_{CBPS} (see Table 11). The interleaver consists of two different permutations as described in Section B3. The value of s is derived from the number of

coded bits per subcarrier, N_{BPSC}, according to $s = \max\left(\frac{N_{BPSC}}{2}, 1\right)$. Note that this process is carried out for N_{BLOCK} (N_{SYM}) iterations. The functional flow is shown in Figure 35.

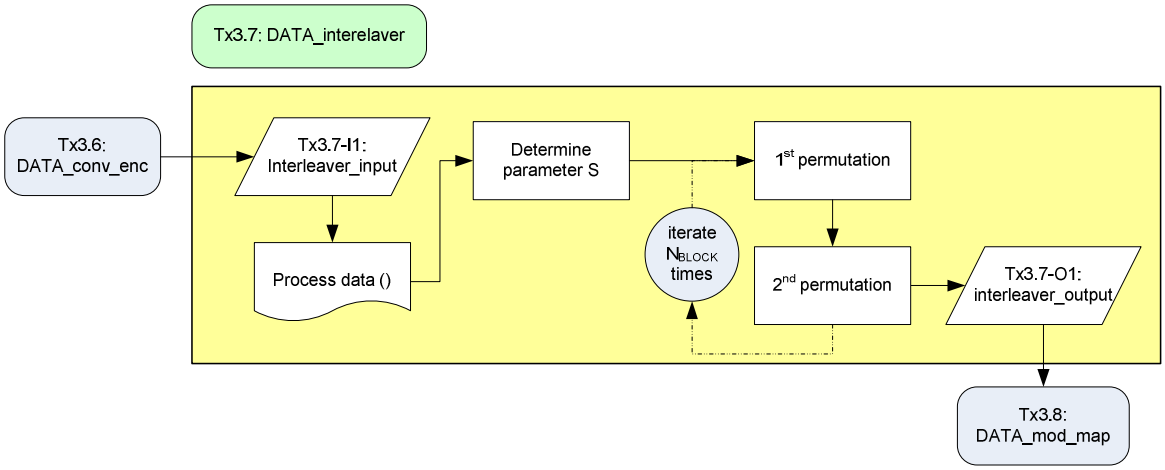

Figure 35. *DATA_interleaver* port and functional flow.

6. Tx3.8: Modulation Mapping (DATA)

Component name: ***DATA_mod_map***

Port design: *DATA_mod_map* has 1 input port and 1 output port. Figure 37 shows the I/O distribution of the component.

Functional design: The DATA OFDM subcarriers is modulated by either BPSK, QPSK, 16-QAM, or 64-QAM modulation, depending on the RATE field. The input binary data shall be converted into complex constellation points as shown in Figure 36. The output values are formed by multiplying the resulting complex value by a normalization factor K_{MOD} (see Table 12) to achieve the same average power for all mappings. The functional flow is shown in Figure 37.

Modulation	K_{MOD}
BPSK	1
QPSK	1/√2
16 QAM	1/√10
64 QAM	1/√42

Table 12. Normalization factor (after: reference [3], Table 81).

	Input bits (b_0)	I-out	Q-out
BPSK	0	-1	0
	1	1	0

	Input bits (b_0)	I-out	Input bits (b_1)	Q-out
QPSK	0	-1	0	-1
	1	1	1	1

	Input bits (b_0,b_1)	I-out	Input bits (b_2,b_3)	Q-out
16 QAM	00	-3	00	-3
	01	-1	01	-1
	11	1	11	1
	10	3	10	3

	Input bits (b_0,b_1,b_2)	I-out	Input bits (b_3,b_4,b_5)	Q-out
64 QAM	000	-7	000	-7
	001	-5	001	-5
	011	-3	011	-3
	010	-1	010	-1
	110	1	110	1
	111	3	111	3
	101	5	101	5
	100	7	100	7

Figure 36. Constellation modulation mapping (after: reference [3], Table 82 to 85).

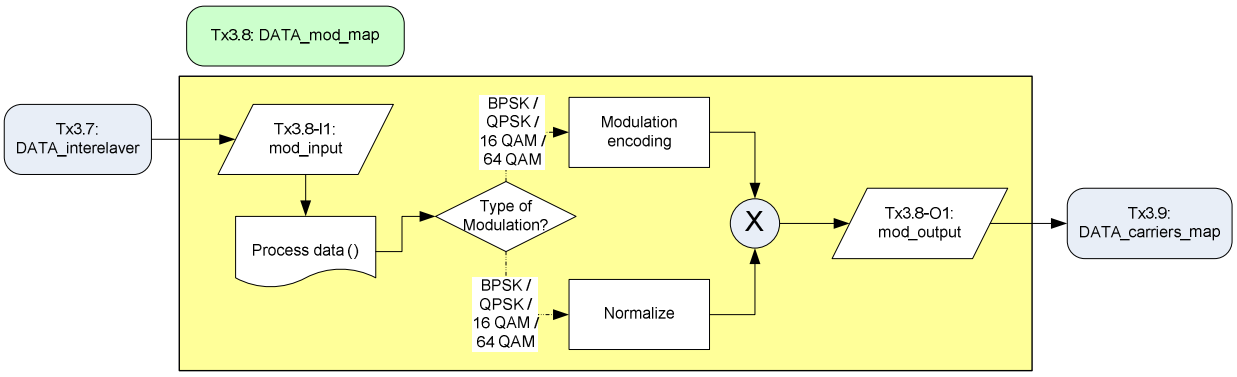

Figure 37. *DATA_mod_map* port and functional flow.

7. Tx3.9: Carriers Mapping (DATA)

Component name: ***DATA_carrier_map***

Port design: *DATA_carrier_map* has 1 input port and 1 output port. Figure 38 shows the I/O distribution of the component.

Functional design: Four pilot tones are inserted to form 52 subcarriers. Similar to its SIGNAL counterpart, *DATA_carrier_map* re-indexes the 52 subcarriers after modulation as part of the 64 frequency samples (by introducing guard bands as stipulated in the IEEE standard) to serve as input into the IFFT component. Note that this process is carried out for N_{SYM} iterations. The functional flow is shown in Figure 38.

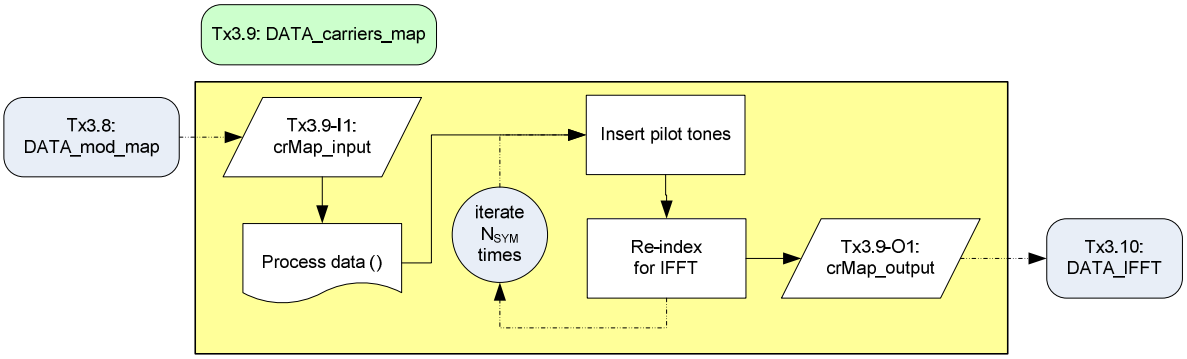

Figure 38. *DATA_carriers_map* port and functional flow.

8. Tx3.10: IFFT (DATA)

Component name: ***DATA_IFFT***

Port design: *DATA_IFFT* has 1 input port and 1 output port. Figure 39 shows the I/O distribution of the component.

Functional design: From *DATA_carriers_map*, 64 frequency samples will be sent through *DATA_IFFT* to convert to 64 time samples. The DIT PINO IFFT algorithm shall be described in Chapter V. Note that this process is carried out for N_{SYM} iterations. The functional flow is shown in Figure 39.

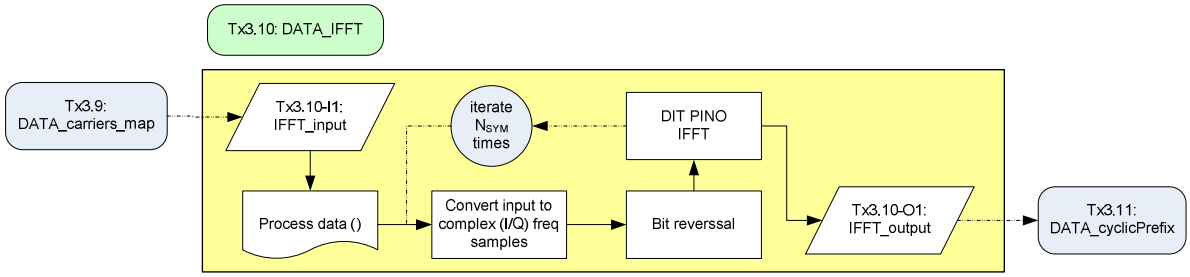

Figure 39. *DATA_IFFT* port and functional flow.

9. Tx3.11: Cyclic Prefix (DATA)

Component name: ***DATA_cyclicPrefix***

Port design: *DATA_cyclicPrefix* has 1 input port and 1 output port. Figure 40 shows the I/O distribution of the component.

Functional design: *DATA_cyclicPrefix* prefixes a quarter of the length of the IFFT time samples to the data to form the DATA subframe. Note that this process is

carried out for N_{SYM} iterations. The final sequence shall be forwarded to *DATA_map* to be concatenated as part of the PPDU. The functional flow is shown in Figure 40.

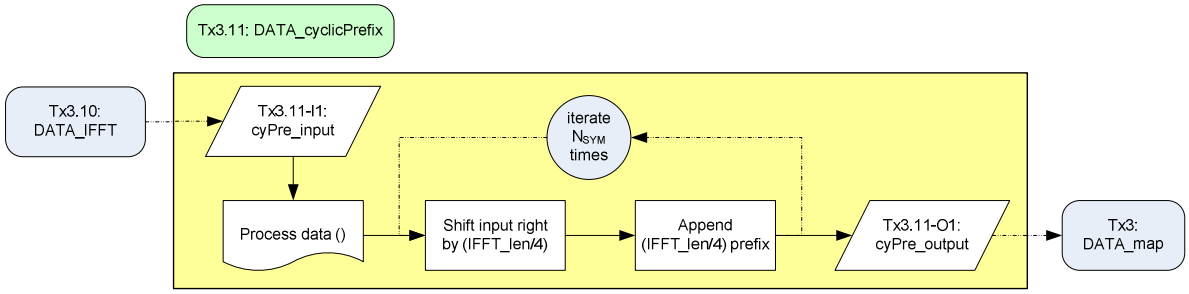

Figure 40. *DATA_cyclicPrefix* port and functional flow.

D. PPDU (FINAL CONCATENATION)

The final piece to the transmitter is the function of concatenating the preamble, SIGNAL and DATA subframes together and form the PPDU frame. This main control is carried out by the *PPDU_map* component.

1. Tx12: PPDU Mapping

Component name: ***PPDU_map***

Port design: *PPDU_map* has a total of 3 input ports and 3 output ports. Figure 41 shows the I/O distribution of the component.

Functional design: As mentioned, *PPDU_map* is the final component that concatenates all the three subframes. It carries out three main functions: (1) retrieves preamble subframe from *preamble_map* component, (2) initiate SIGNAL processing and storage and (3) initiate DATA processing and storage. The final PPDU frame is ready to be sent for hardware DAC and RF up-conversion to the 5GHz range before transmission as stipulated in the IEEE 802.11a standard. The functional flow is shown in Figure 41.

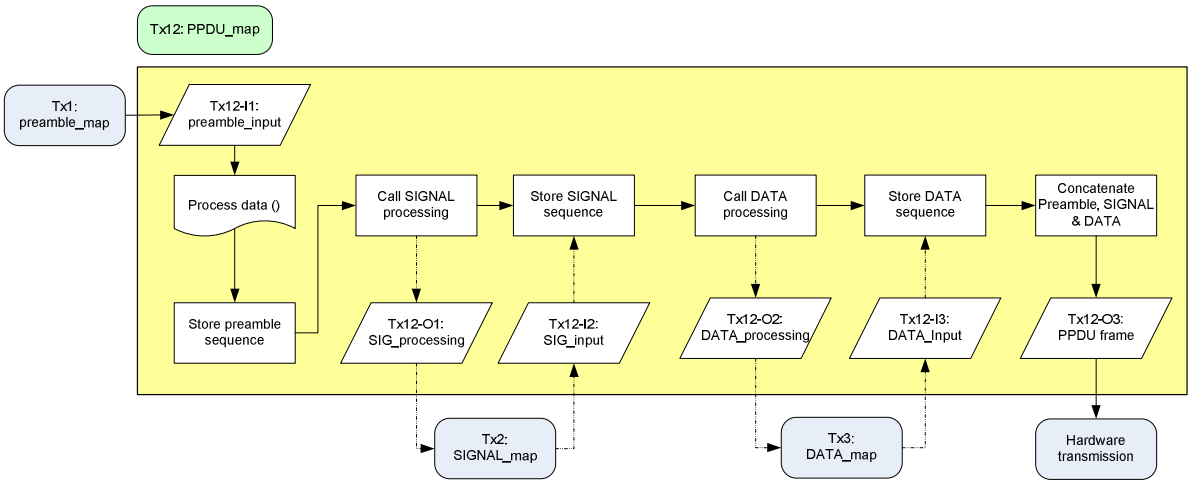

Figure 41. *PPDU_map* port and functional flow.

In this chapter, the developmental details of the transmitter to model the IEEE 802.11a PHY layer have been described. The transmitter components consist of the preamble, SIGNAL and DATA subframes. These three subframes are concatenated subsequently to from the PPDU frame. In the next chapter, we shall focus on describing the developmental details for the receiver components.

IV. DEVELOP: RECEIVER

This chapter focuses on the receiver portion of the IEEE 802.11a PHY layer. Like before, the components are described according to the inter-linkages of the input-output (I/O) ports and the functional implementation in C++ code.

The receiver converts the digitized PPDU frames (passed down from the ADC after down-conversion from the RF front end) into binary outputs from which the original PSDU information can be extracted. The incremental development is discussed here starting with the removal of the preamble subframe, follow by the SIGNAL subframe, and finally, the DATA subframe where the PSDU is extracted.

A. PREAMBLE

In this subframe, the two core tasks are to receive the digitized time samples and remove the preamble subframe prior further processing. The preamble subframe consists of training sequences, which are predictable such that it can be tracked and removed from the received data. Table 13 summarizes the components needed to remove the preamble subframe from the received data.

Index	Component	Functions
Rx0	**Rx_data**	- receive digitized data stream
Rx1 / Rx12	**PPDU_rx**	- extract the required digitized PPDU stream - removed preamble from PPDU - send stream for SIG removal

Table 13. IEEE 802.11a Receiver preamble subframe components functionalities.

1. Rx0: Receiver Data (Assembly Controller)

Component name: ***Rx_data***

Port design: *Rx_data* is the assembly controller for the receiver. It initiates the receiver software routine to extract the PSDU frame. It has one output port that sends continuous digitized time samples to *PPDU_rx* component. Figure 42 shows the I/O distribution of the component.

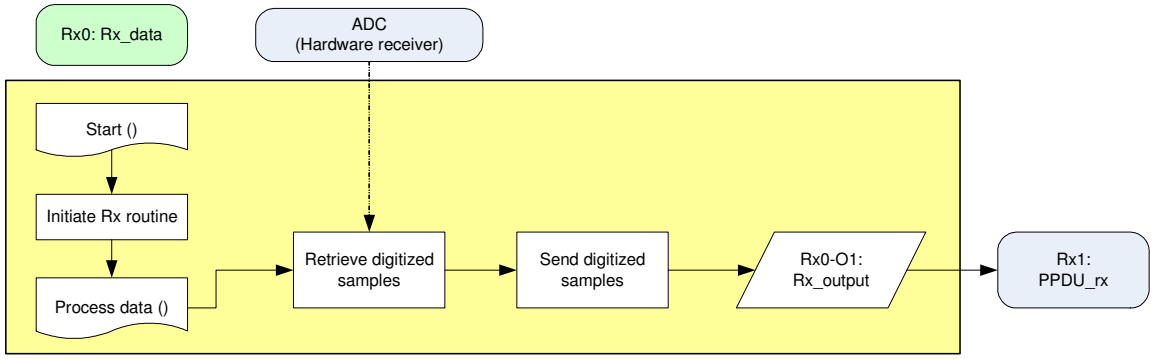

Figure 42. *Rx_data* port and functional flow.

Functional design: *Rx_data* emulates the ADC interface at the receiver by sending digitized samples for software processing. For testing purposes, the test data provided in Annex G of IEEE 802.11a standard [3] shall be sent out from *Rx_data* one sample at a time. It will be followed by a stream of 'zero' until it reaches the upper limit assumed in the simulation. This sequential functional flow is shown in Figure 42.

2. Rx1: PPDU Receiver

Component name: ***PPDU_rx***

Port design: *PPDU_rx* has a total of 1 input port and 1 output port. Figure 43 shows the I/O distribution of the component.

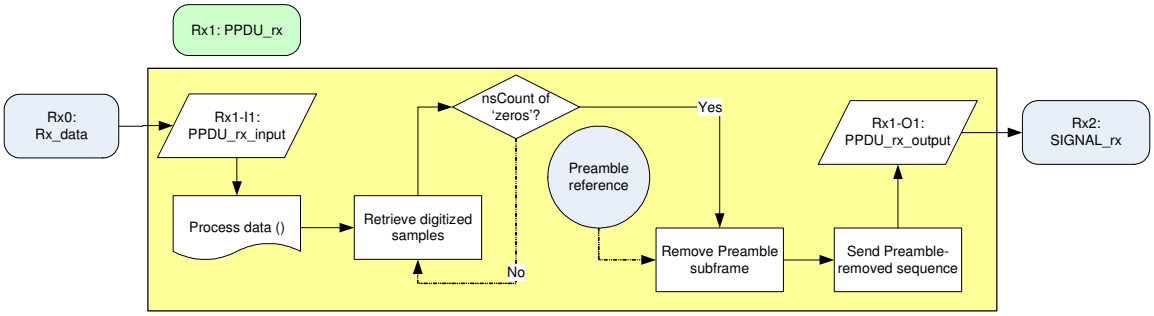

Figure 43. *PPDU_rx* port and functional flow.

Functional design: *PPDU_rx* carries out three main functions: (1) receive the digitized time samples, (2) remove preamble subframe, and (3) send the processed data stream for SIGNAL subframe processing. For synchronization, the digitized samples are continually received until the component senses a continuous stream of 'zeros' (determine by *nsCount*). This indicates all the digitized samples have been received. The received data stream is compared with the preamble training sequence (a concatenation of ST and LT sequences) that is specified in the IEEE 802.11a standard [3] and, therefore,

known in advance, and the remaining (SIGNAL + DATA) subframes are extracted. The functional flow is shown in Figure 43.

B. SIGNAL

The SIGNAL symbol consists of the critical RATE and LENGTH fields that are encoded by a convolutional code of $R = \frac{1}{2}$. In order to remove this subframe, the functionalities described in the equivalent transmitter subframe shall be reversed to extract the two critical fields. The decoding of the SIGNAL field into an OFDM symbol follows the procedures: removing the cyclic prefix, FFT, removing the pilot tones (guard bands), BPSK demodulation, deinterleaving and, finally, convolutional decoding at a data receiver rate of 6 Mbits/s. Table 14 summarizes the components needed to extract the necessary fields.

Index	Component	Functions
Rx2	**Header_rx (SIGNAL_rx)**	- removed SIG from PPDU - send SIG for processing - extract RATE & LENGTH from SIG - send received data for processing
Rx2.11	**SIG_cyclicprefix_rem**	- SIG cyclic prefix removal
Rx2.10	**SIG_FFT**	- SIG FFT
Rx2.9	**SIG_carriers_demap**	- SIG carriers demapping
Rx2.8	**SIG_BPSK_demod**	- SIG BPSK demodulation
Rx2.7	**SIG_deinterleaver**	- SIG deinterleaving
Rx2.6	**SIG_conv_dec**	- SIG convolutional decoding

Table 14. IEEE 802.11a Receiver SIGNAL subframe components functionalities.

1. Rx2: SIGNAL Receiver

Component name: ***SIGNAL_rx***

Port design: *SIGNAL_rx* has a total of 2 input ports and 2 output ports. Figure 44 shows the I/O distribution of the component.

Functional design: *SIGNAL_rx* carries out the function of extracting important parameters (RATE and LENGTH) and sends them for DATA processing together with the DATA subframe. At a data rate of 6 Mbits/s, the modulation type and coding parameters are fixed as shown in Table 15. The functional flow is shown in Figure 44.

Data rate $(Mbits/s)$	Modulation	Coding rate (R)	Coded bits Per subcarrier (N_{BPSC})	Coded bits per OFDM symbol (N_{CBPS})	Data bits per OFDM symbol (N_{DBPS})
6	BPSK	1/2	1	48	24

Table 15. Rate-dependent parameters: 6 Mbits/s.

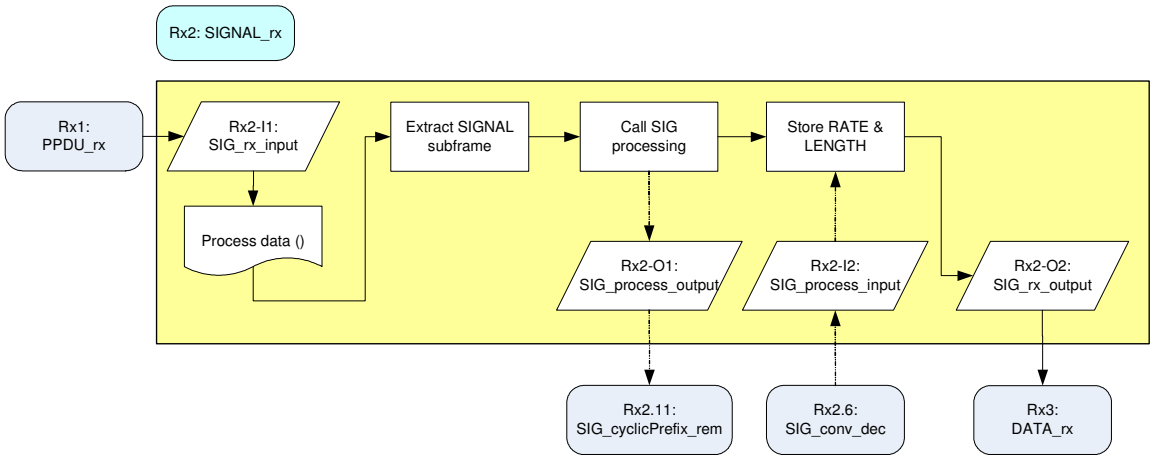

Figure 44. *SIGNAL_rx* port and functional flow.

2. Rx2.11: Cyclic Prefix Removal (SIG)

Component name: ***SIG_cyclicPrefix_rem***

Port design: *SIG_cyclicPrefix_rem* has a total of 1 input port and 1 output port. Figure 45 shows the I/O distribution of the component.

Functional design: *SIG_cyclicPrefix_rem* removes the prefixes (a quarter of the length of the IFFT time samples) from the input data to retrieve the original time samples obtained from the transmitter IFFT. This sequence shall be forwarded to *SIG_FFT* for the FFT. The functional flow is shown in Figure 45.

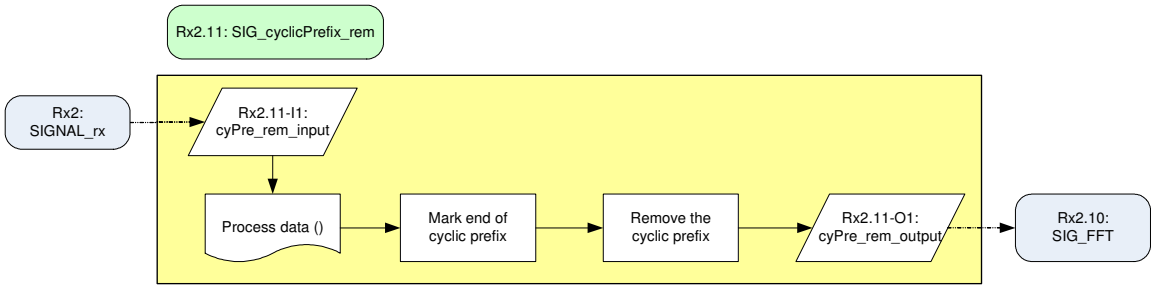

Figure 45. *SIG_cyclicPrefix_rem* port and functional flow.

3. Rx2.10: FFT (SIG)

Component name: ***SIG_FFT***

Port design: *SIG_FFT* has 1 input port and 1 output port. Figure 46 shows the I/O distribution of the component.

Functional design: From *SIG_cyclicPrefix_rem*, 64 time samples will be sent through *SIG_FFT* to convert to 64 frequency samples. The DIT PINO FFT algorithm shall be described in Chapter V. The functional flow is shown in Figure 46.

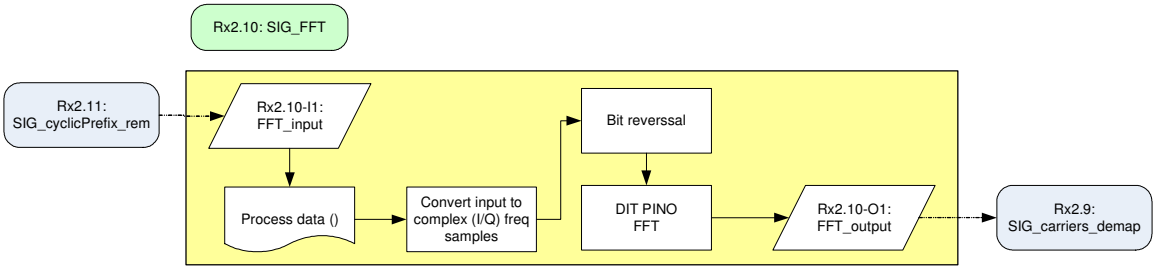

Figure 46. *SIG_FFT* port and functional flow.

4. Rx2.9: Carriers Demapper (SIG)

Component name: ***SIG_carriers_demap***

Port design: *SIG_carriers_demap* has 1 input port and 1 output port. Figure 47 shows the I/O distribution of the component.

Functional design: *SIG_carriers_demap* re-indexes the 64 frequency samples (by removing guard bands as stipulated in the IEEE standard) and retrieves the 52 subcarriers. Four pilot tones are removed before passing the data stream for BPSK demodulation. The functional flow is shown in Figure 47.

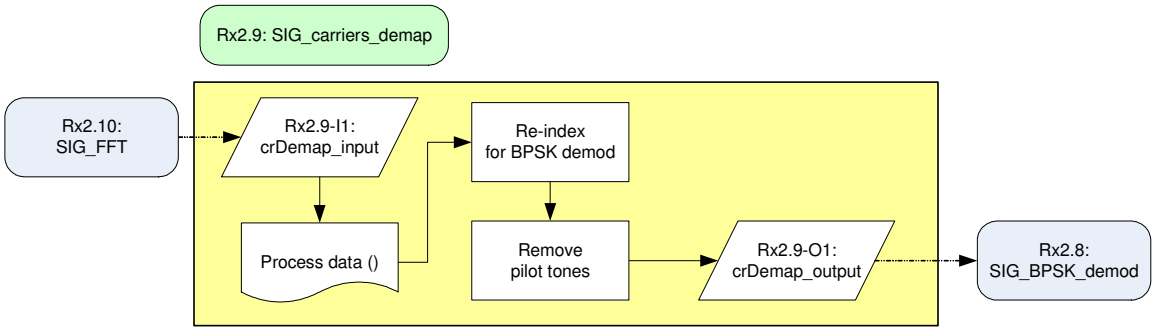

Figure 47. *SIG_carriers_demap* port and functional flow.

5. Rx2.8: BPSK Demodulator (SIG)

Component name: ***SIG_BPSK_demod***

Port design: *SIG_BPSK demod* has 1 input port and 1 output port. Figure 48 shows the I/O distribution of the component.

Functional design: The SIGNAL bits are retrieved by using BPSK demodulation. The 48 frequency subcarriers are demodulated to retrieve the encoded and interleaved binary data, taking into consideration the normalization factor, K_{MOD} (see Table 12). For BPSK demodulation, K_{MOD} is unity. The functional flow is shown in Figure 48.

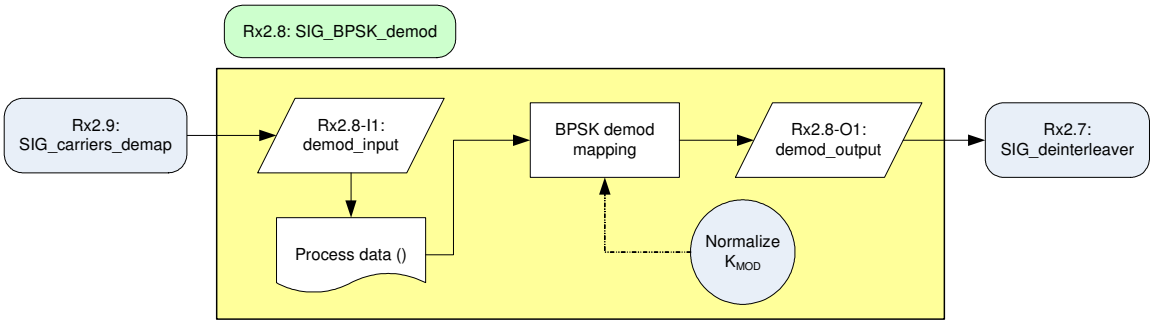

Figure 48. *SIG_BPSK demod* port and functional flow.

6. Rx2.7: De-Interleaver (SIG)

Component name: ***SIG_deinterleaver***

Port design: *SIG_deinterleaver* has 1 input port and 1 output port. Figure 49 shows the I/O distribution of the component.

Functional design: All data bits are passed through a block deinterleaver after demodulation. The deinterleaver has a block size equals to the number of coded bits in a single OFDM symbol, N_{CBPS} (see Table 15). The deinterleaver consists of two different permutations, which is the inverse of the transmitter interleaver described in Chapter III. Note that j, i and k refer to the index of the coded bit before the first, before the second and after the second permutation, respectively. Note that $s = \max\left(\frac{N_{BPSC}}{2}, 1\right)$. Equation 6 and 7 respectively describe the first and second permutations.

$$i = s \times floor\left(\frac{j}{s}\right) + \left(j + floor(16 \times \frac{j}{N_{CBPS}})\right) \bmod s \quad i, j, k = 0,1,\ldots,N_{CBPS}\text{-}1 \tag{6}$$

$$k = 16 \times i - (N_{CBPS} - 1)\, floor(16 \times \frac{i}{N_{CBPS}}) \quad (7)$$

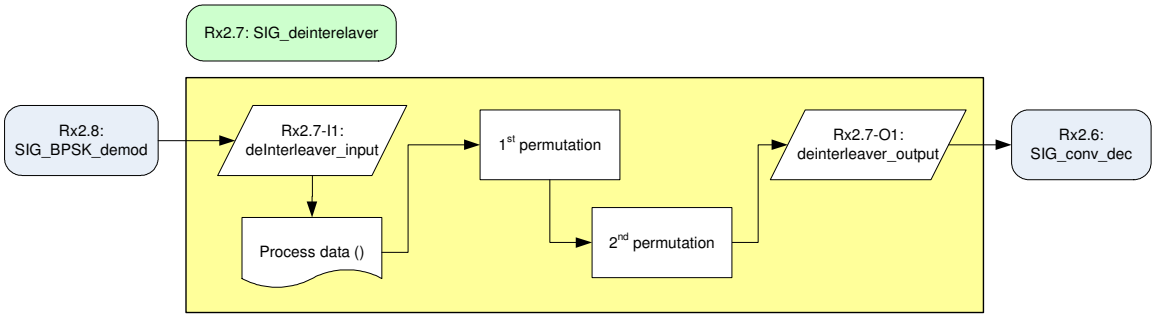

Figure 49. *SIG_deinterleaver* port and functional flow.

7. Rx2.6: Convolutional Decoder (SIG)

Component name: ***SIG_conv_dec***

Port design: *SIG_conv_dec* has 1 input port and 1 output port. Figure 50 shows the I/O distribution of the component.

Functional design: Viterbi decoding is chosen to decode the stream of convolutional bits. Since the SIGNAL fields have been coded with a convolutional encoder of coding rate $R = \frac{1}{2}$, there is no need to insert dummy bits prior decoding. Details of the Viterbi decoder algorithm are presented in Chapter V. The functional flow is shown in Figure 50.

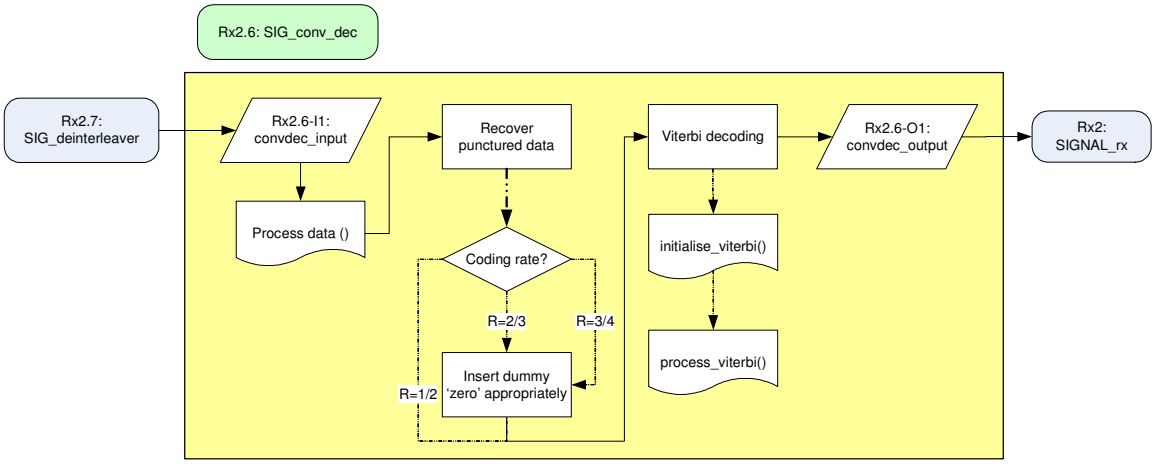

Figure 50. *SIG_conv_dec* port and functional flow.

C. DATA

Besides all the components (functionalities) under the SIGNAL subframe, extracting the PSDU from the DATA subframe requires an additional procedure of descrambling the decoded bits. Table 16 summarizes the components needed to extract the PSDU.

Index	Component	Functions
Rx3	**data_rx**	- receive and send raw data for processing - receive and send PSDU data to MAC layer
Rx3.11	**data_cyclicprefix_rem**	- data cyclic prefix removal
Rx3.10	**data_FFT**	- data FFT
Rx3.9	**data_carriers_demap**	- data carriers demapping
Rx3.8	**data_demod_map**	- data demodulation mapping
Rx3.7	**data_deinterleaver**	- data deinterleaving
Rx3.6	**data_conv_dec**	- data dummy insertion - data convolutional decoding
Rx3.5	**data_tail_replace**	- not required, encompass in descrambler
Rx3.4	**data_descrambler**	- descramble the raw data

Table 16. IEEE 802.11a Receiver DATA subframe components functionalities.

1. Rx3: DATA Receiver

Component name: ***DATA_rx***

Port design: *DATA_rx* has a total of 2 input ports and 2 output ports. Figure 51 shows the I/O distribution of the component.

Functional design: *DATA_rx* is the heart of DATA processing at the PHY layer where the PSDU are extracted. This is possible by referencing information provided by RATE and LENGTH fields. From RATE, the modulation type and coding parameters are determined according to Table 17. The LENGTH field determines the size of PSDU in the DATA. The functional flow is shown in Figure 51.

Data rate $(Mbits/s)$	Modulation	Coding rate (R)	Coded bits per subcarrier (N_{BPSC})	Coded bits per OFDM symbol (N_{CBPS})	Data bits per OFDM symbol (N_{DBPS})
6	BPSK	1/2	1	48	24
9	BPSK	3/4	1	48	36
12	QPSK	1/2	2	96	48
18	QPSK	3/4	2	96	72
24	16-QAM	1/2	4	192	96
36	16-QAM	3/4	4	192	144
48	64-QAM	2/3	6	288	192
54	64-QAM	3/4	6	288	216

Table 17. Rate-dependent parameters.

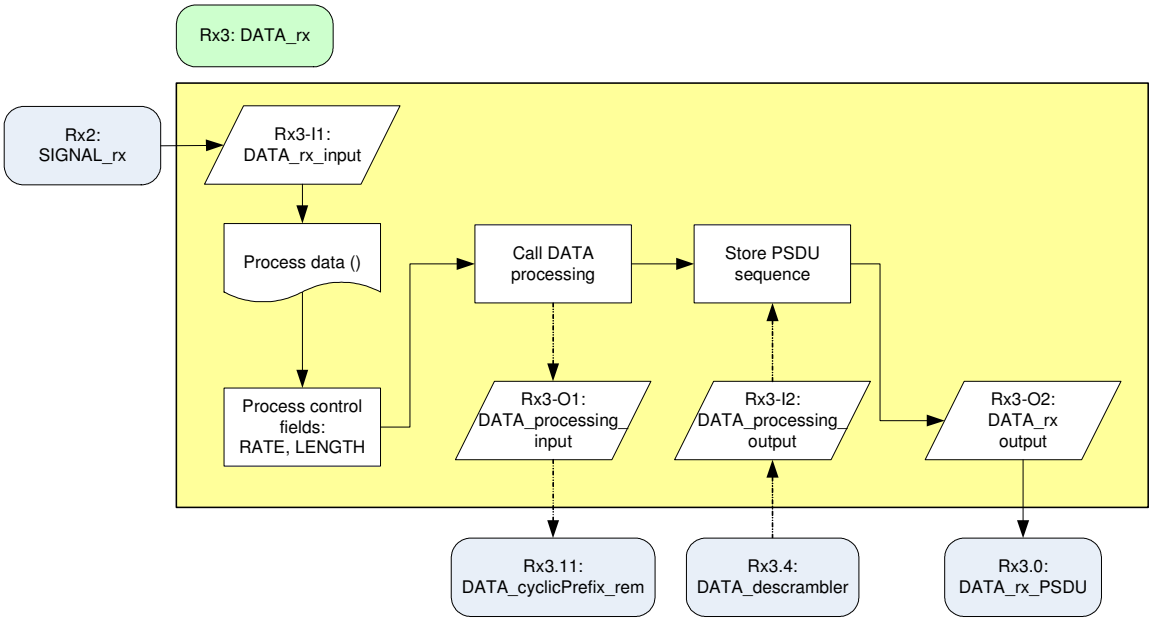

Figure 51. *DATA_map* port and functional flow.

2. Rx3.11: Cyclic Prefix Removal (DATA)

Component name: ***DATA_cyclicPrefix_rem***

Port design: *DATA_cyclicPrefix_rem* has a total of 1 input port and 1 output port. Figure 52 shows the I/O distribution of the component.

Functional design: *DATA_cyclicPrefix_rem* removes the prefixes (a quarter of the length of the IFFT time samples) from the input data to retrieve the original time samples obtained from the transmitter IFFT. Note that this process is carried out for N_{SYM} iterations. This sequence shall be forwarded to *DATA_FFT* component for the FFT. The functional flow is shown in Figure 52.

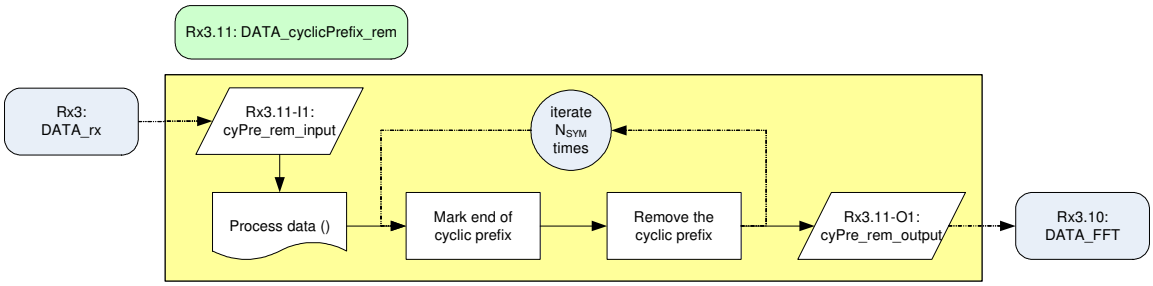

Figure 52. *DATA_cyclicPrefix_rem* port and functional flow.

3. Rx3.10: FFT (DATA)

Component name: ***DATA_ FFT***

Port design: *DATA_FFT* has 1 input port and 1 output port. Figure 53 shows the I/O distribution of the component.

Functional design: From *DATA_ cyclicPrefix_rem*, every 64 time samples will be sent through *DATA_FFT* to convert to 64 frequency samples. The DIT PINO FFT algorithm shall be described under Chapter V. Note that this process is carried out for N_{SYM} iterations. The functional flow is shown in Figure 53.

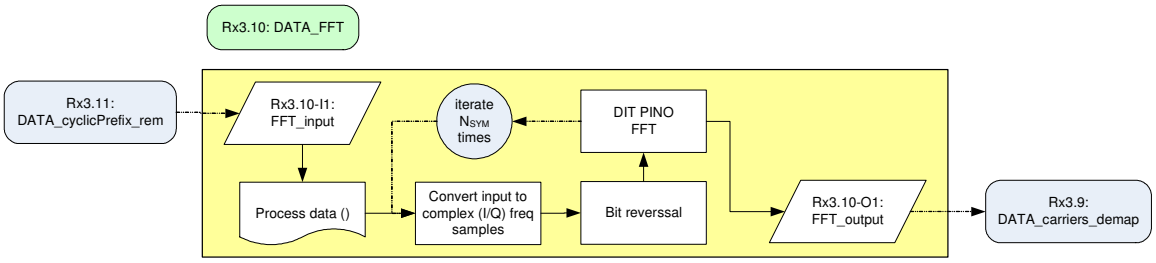

Figure 53. *DATA_FFT* port and functional flow.

4. Rx3.9: Carriers Demapper (DATA)

Component name: ***DATA_carriers_demap***

Port design: *DATA_carriers_demap* has 1 input port and 1 output port. Figure 54 shows the I/O distribution of the component.

Functional design: *DATA_carriers_demap* re-indexes every 64 frequency samples (by removing guard bands) and retrieves the 52 subcarriers. Four pilot tones are also removed for demodulation mapping. Note that this process is carried out for N_{SYM} iterations. The functional flow is shown in Figure 54.

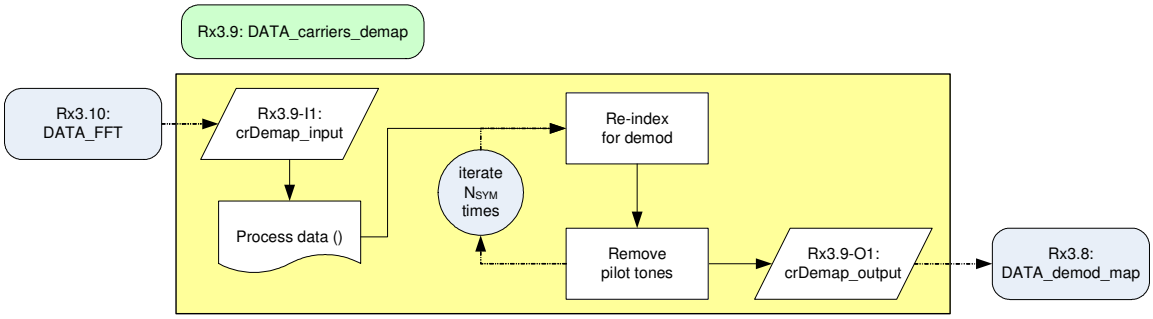

Figure 54. *DATA_carriers_demap* port and functional flow.

5. Rx3.8: Demodulation Mapper (DATA)

Component name: ***DATA_demod_map***

Port design: *DATA_demod_map* has 1 input port and 1 output port. Figure 56 shows the I/O distribution of the component.

Functional design: The DATA OFDM subcarriers are demodulated by using BPSK, QPSK, 16-QAM, or 64-QAM, depending on the RATE field. The gray-coded complex constellation points shall be converted to binary input data according to Figure 55. Note that the normalization factor K_{MOD} (see Table 12) changes with modulation scheme to ensure the same average power is achieved for all mappings. The functional flow is shown in Figure 56.

	I-in	Output bits (b_0)
BPSK	-1	0
	1	1

	I-in	Output bits (b_0)	Q-in	Output bits (b_1)
QPSK	-1	0	-1	0
	1	1	1	1

	I-in	Output bits (b_0,b_1)	Q-in	Output bits (b_2,b_3)
16 QAM	-3	00	-3	00
	-1	01	-1	01
	1	11	1	11
	3	10	3	10

	I-in	Output bits (b_0,b_1,b_2)	Q-in	Output bits (b_3,b_4,b_5)
64 QAM	-7	000	-7	000
	-5	001	-5	001
	-3	011	-3	011
	-1	010	-1	010
	1	110	1	110
	3	111	3	111
	5	101	5	101
	7	100	7	100

Figure 55. Constellation demodulation mapping.

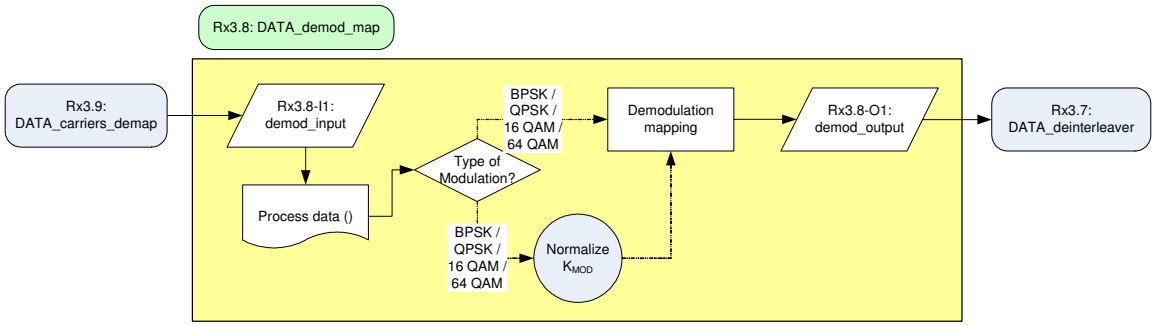

Figure 56. *DATA_demod_map* port and functional flow.

6. Rx3.7: De-Interleaver (DATA)

Component name: ***DATA_deinterleaver***

Port design: *DATA_deinterleaver* has 1 input port and 1 output port. Figure 57 shows the I/O distribution of the component.

Functional design: All data bits are passed through a block deinterleaver after demodulation. The deinterleaver has a block size equals to the number of coded bits in a single OFDM symbol, N_{CBPS} (see Table 17). The deinterleaver consists of two different permutations as described in Section B6. The value s is determined by the number of coded bits per subcarrier, N_{BPSC}, whereby $s = \max\left(\frac{N_{BPSC}}{2}, 1\right)$. Note that this process is carried out for N_{BLOCK} (N_{SYM}) iterations. The functional flow is shown in Figure 57.

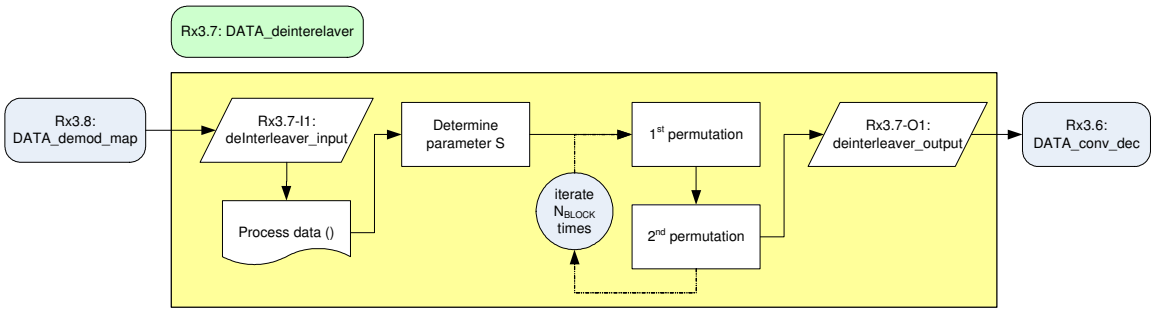

Figure 57. *DATA_deinterleaver* port and functional flow.

7. Rx3.6: Convolutional Decoder (DATA)

Component name: ***DATA_conv_dec***

Port design: *DATA_conv_dec* has 1 input port and 1 output port. Figure 59 shows the I/O distribution of the component.

Functional design: Viterbi decoding is chosen to decode the stream of convolutional bits. The DATA bits have been coded with a convolutional encoder of coding rate $R = \frac{1}{2}$, $\frac{2}{3}$, or $\frac{3}{4}$, depending on the data rate (see Table 17). Since higher rates of $R = \frac{2}{3}$ and $\frac{3}{4}$ are derived from $R = \frac{1}{2}$ by employing puncturing at the transmitter, conversely, at the receiver, dummy bits have to be inserted prior to the

decoding. There is no need to insert dummy bits prior to decoding for $R=\frac{1}{2}$. The dummy bits insertion patterns are illustrated in Figure 58 for $R=\frac{2}{3}$ and $\frac{3}{4}$. Details of the Viterbi decoder algorithm is presented in Chapter V. Note that *process_viterbi()* process call is carried out for N_{SYM} iterations. The functional flow is shown in Figure 59.

Insertion pattern: R=3/4

Received bits	A0	B0	A1	B2	A3	B3	A4	B5	A6	B6	A7	B8
Dummy bits	A0	A1	A2	A3	A4	A5	A6	A7	A8			
	B0	B1	B2	B3	B4	B5	B6	B7	B8			
Decoded data	y0	y1	y2	y3	y4	y5	y6	y7	y8			

Insertion pattern: R=2/3

Received bits	A0	B0	A1	A2	B2	A3	A4	B4	A5	A6	B6	A7	A8	B8	A9
Dummy bits	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9					
	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9					
Decoded data	y0	y1	y2	y3	y4	y5	y6	y7	y8	y9					

Figure 58. *DATA_conv_dec* puncturing patterns.

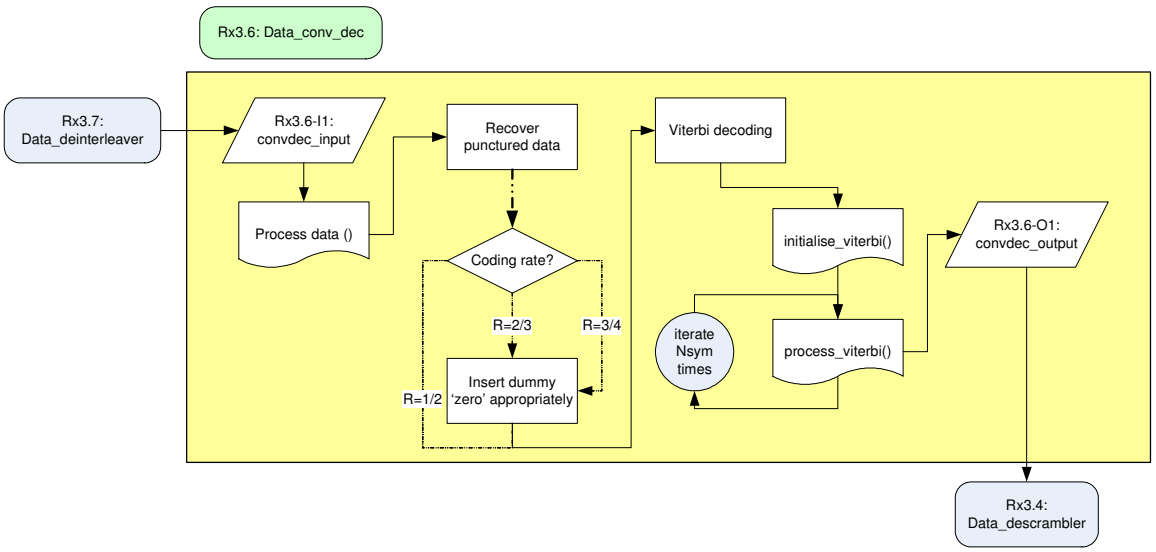

Figure 59. *DATA_conv_dec* port and functional flow.

8. Rx3.5: Tail Replacement (DATA)

This function is subsumed under the descrambler component below.

9. Rx3.4: Descrambler (DATA)

Component name: ***DATA_descrambler***

Port design: *DATA_descrambler* has 1 input port and 1 output port. Figure 61 shows the I/O distribution of the component.

Functional design: The DATA subframe shall be descrambled by passing through the same length-127 scrambler as illustrated in Figure 60. The scrambler is set to a pseudo random non-zero initial state when first applied to the input data. This must be the same as that being used in the transmitter scrambler to ensure the descrambling process is synchronized. For the simulation and testing in this thesis research, this pseudo-random non-zero state has been set to "1011101" as proposed in the test case under Annex G of IEEE 802.11a standard [3]. The functional flow is shown in Figure 61.

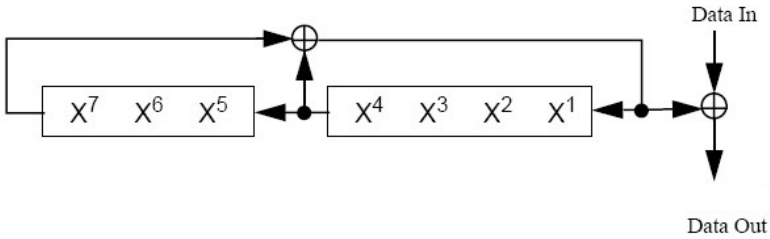

Figure 60. DATA scrambler.

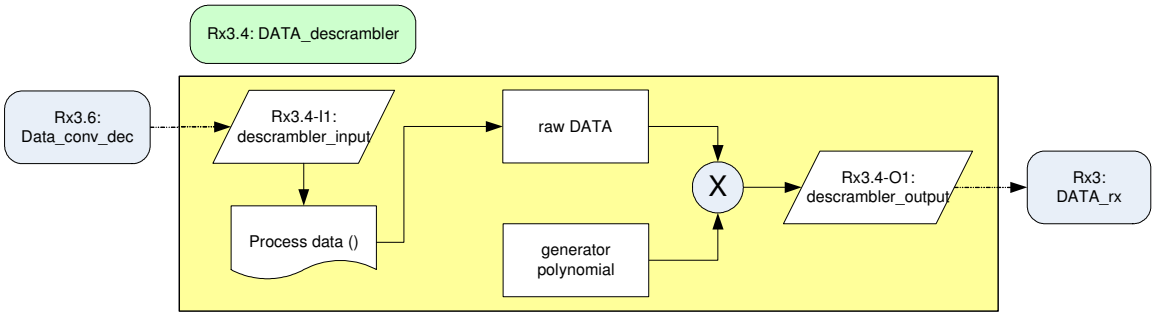

Figure 61. *DATA_descrambler* port and functional flow.

With an appreciation of the development of the transceiver, the next chapter shall proceed to touch on the challenges that were experienced during the conduct of the thesis research.

V. CHALLENGES

This chapter focuses on the major challenges that were encountered while developing the IEEE 802.11a PHY layer model. In terms of developing the functionalities, two complex functions are singled out specifically for discussion. They are the (inverse) fast Fourier transform (IFFT/FFT) and the Viterbi decoder as described in Section A below. Being developmental software, OSSIE is still considered in its early stage of maturity. While the current version is useful and sufficient for modeling the standard, enhancement will definitely help in fine-tuning and optimizing the performance of the model. These software and integration challenges are discussed in Section B.

A. SPECIAL INTEREST COMPONENTS

The FFT and IFFT are software algorithms for (inverse) discrete Fourier transform (DFT/IDFT) that are used to emulate the OFDM capabilities in the standard. OFDM advocates parallel data transmission scheme that reduces the effect of multipath fading and prevent the need of complex equalizers at the receiver. The mathematical details can be obtained from reference [6]. For convolutional decoding, the Viterbi algorithm is recommended by the IEEE 802.11a standard [3]. Using the concept of a trellis representation, hard decision decoding with minimum Hamming distance selection is implemented. Both of the above functions follow closely to the explanation provided in [7] and are described here.

1. IFFT / FFT

An OFDM transmission system typically consists of three stages each for both the transmitter and the receiver. This is illustrated in Figure 62. In the transmitter, the serial-to-parallel (s/p) converter shall prepare a stack of 64 frequency samples (including pilot tones and guard bands) for the IFFT. Sixty-four time samples are generated and cyclic prefix shall be inserted to provide redundancy and to mitigate inter-symbol interference (ISI). Conversely, at the receiver, the cyclic prefix shall be removed prior to the FFT. Sixty-four time samples are sent through the FFT to obtain 64 frequency samples. The

pilot tones and guard bands are removed before passing through the parallel-to-serial (p/s) converter.

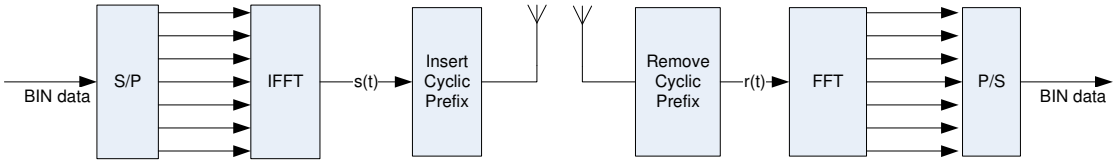

Figure 62. OFDM transmission system: transmitter and receiver.

For a discussion on the DFT, see Chapter 5 of [7]. Two possible types of algorithms for DFTs are decimation-in-time (DIT) and decimation-in-frequency (DIF). DIT transforms are based on splitting the DFT into two summations: (1) a decimated time sequence where even-indexed samples are removed and (2) a decimated time sequence where odd-indexed samples are removed. Similarly, DIF transforms split the DFT summation into two summations: (1) a decimated frequency sequence where even-indexed samples are removed and (2) a decimated frequency sequence where odd-indexed samples are removed. For the IEEE 802.11a implementation in this research, the DIT approach is chosen.

If the inputs to either the IFFT or FFT are in natural order, the outputs shall turn out to be in bit-reversed order. This is known as natural-input-permutated-output (NIPO). Conversely, if the inputs to the transform are in bit-reversed order, the outputs shall turn out to be in natural order. This is known as permutated-input-natural-output (PINO). The PINO implementation is chosen for this thesis to provide a natural output sequence.

The above explains the term DIT PINO IFFT/FFT that has been used to describe the DFT components in the model. The function flows for both the *DATA_IFFT* of the transmitter and the *DATA_FFT* of the receiver are shown in Figure 63 for comparisons. It is observed that most of the processes within the components remain the same except that DIT PINO IFFT is called in *DATA_IFFT* while DIT PINO FFT is called in *DATA_FFT*. The components can be described by three main processes: (1) convert inputs to complex constellations of I and Q samples, (2) reverse the bits for PINO operations and (3) carry out either DIT PINO IFFT or DIT PINO FFT.

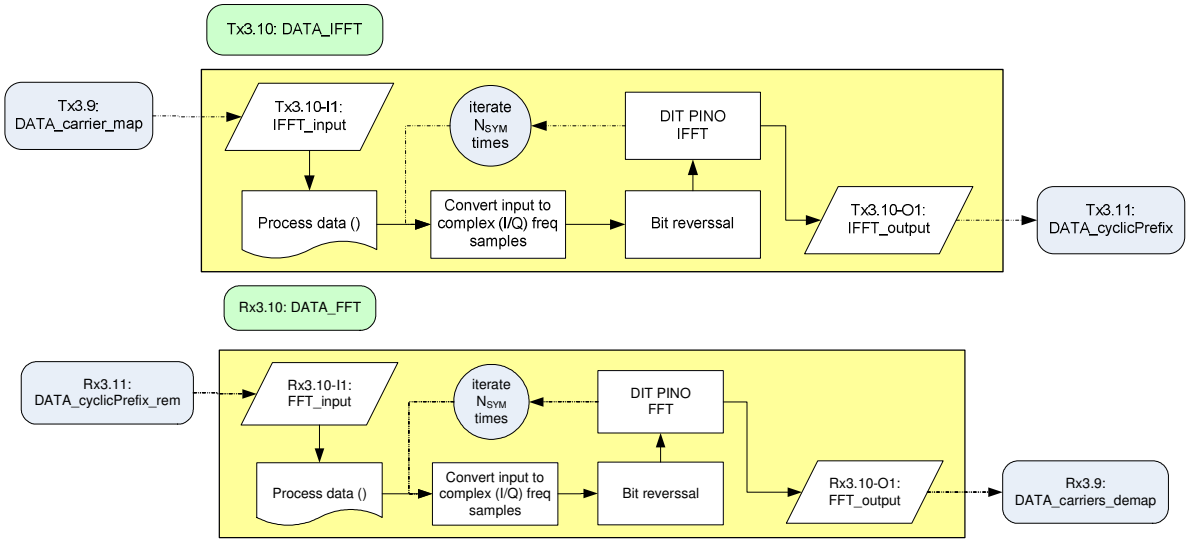

Figure 63. *DATA_IFFT* and *DATA_FFT* functional flows.

a. Real to Complex Conversion

As the inputs to the component are real values, there is a requirement to convert them to complex values prior to the DFT. By default, complex number arithmetic is not supported in the included libraries. The header file: *complex.h* has to be included before any complex algorithm can be built. The I-channel and Q-channel values shall be mapped to the real and imaginary parts of the complex number, respectively.

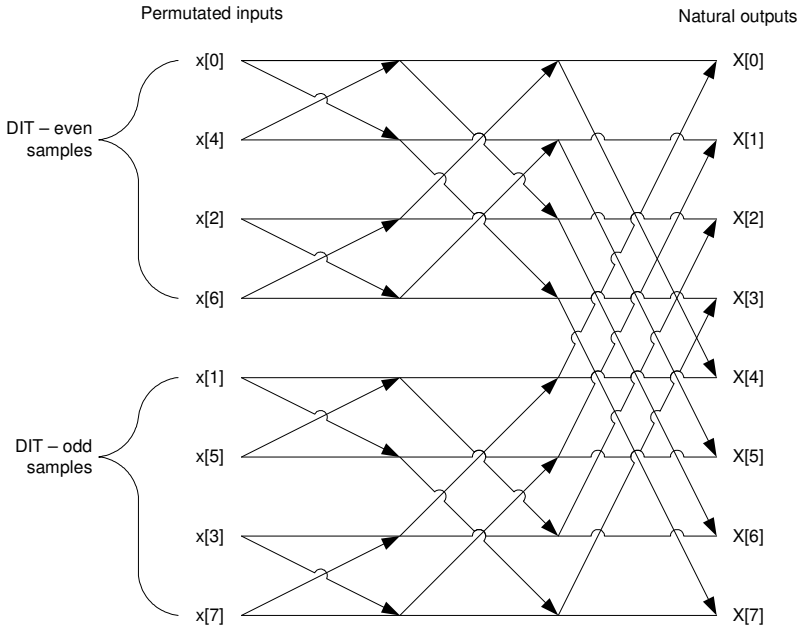

Figure 64. A sample signal flow graph of a DIT PINO FFT.

b. Bit Reversal

As explained earlier, PINO is preferred and input bits have to be bit-reversed prior to IFFT/FFT operations. A sample of the bit-reversed inputs is shown in Figure 64. Notice that the outputs sequence is in natural order due to such arrangement.

c. DIT PINO DFT

From Figure 64, it is observed that the DFT is implemented by individual 'butterfly' patterns. This is the backbone of the IFFT/FFT algorithm. Both the DFT and inverse DFT are defined in Equation 8 and 9 respectively:

$$X[m] = \sum_{n=0}^{N-1} x[n] W_N^{-mn} \tag{8}$$

$$x[n] = \frac{1}{N} \sum_{n=0}^{N-1} X[m] W_N^{mn} \tag{9}$$

While not going into details of the mathematical operations (see [7] for derivations), it is important to realize that both the IFFT and FFT algorithms are almost similar except for the exponent of the W_N function and the normalization factor. The normalization factor for IFFT is equal to the number of samples (64), while that for the FFT is unity. If we let $W_{DFT} = W_N^{-mn}$ and $W_{IDFT} = W_N^{mn}$, it can be shown that $\mathrm{Re}(W_{DFT}) = \mathrm{Re}(W_{IDFT})$ and $\mathrm{Im}(W_{DFT}) = -\mathrm{Im}(W_{IDFT})$. In other words, if θ is the argument, then the two functions differ as follows:

$$\mathrm{W_{DFT}} = \cos(\theta) + j\sin(\theta) \tag{10}$$

$$\mathrm{W_{IDFT}} = \cos(\theta) - j\sin(\theta) \tag{11}$$

2. Viterbi Decoder

The decoder carries out two main functions: (1) inserting the dummy "zero"s and (2) convolutional decoding using the Viterbi algorithm. The DATA subframe has been coded with a convolutional encoder of coding rate $R = \frac{1}{2}, \frac{2}{3}$, or $\frac{3}{4}$, depending on the data rate. The convolutional encoder was described under the transmitter development earlier. To compensate for encoder puncturing, dummy "zero" bits need to be inserted into the convolutional decoder in place of the omitted bits. Decoding by the Viterbi algorithm is preferred, especially for convolutional coding. This sequential functional flow is shown in Figure 65.

a. Initialise_viterbi()

The Viterbi algorithm with hard decision decoding is based on finding the shortest Hamming distance by comparing the received data bits with a set of expected code sequences. A lookup matrix is constructed to assist in the process. For DATA decoding, the lookup matrix is of size 64 (rows) by 22 (columns). Since the constraint length is 7, there are six memory elements in the encoder. This will entail a total of 2^6 = 64 different states in the shift register memories, which translates to 64 rows in the lookup matrix. The compositions of the 22 columns are provided in Table 18. The lookup matrix needs to be initialized prior to the decoding process. This is carried out by the *initialise_viterbi()* function call as shown in Figure 66.

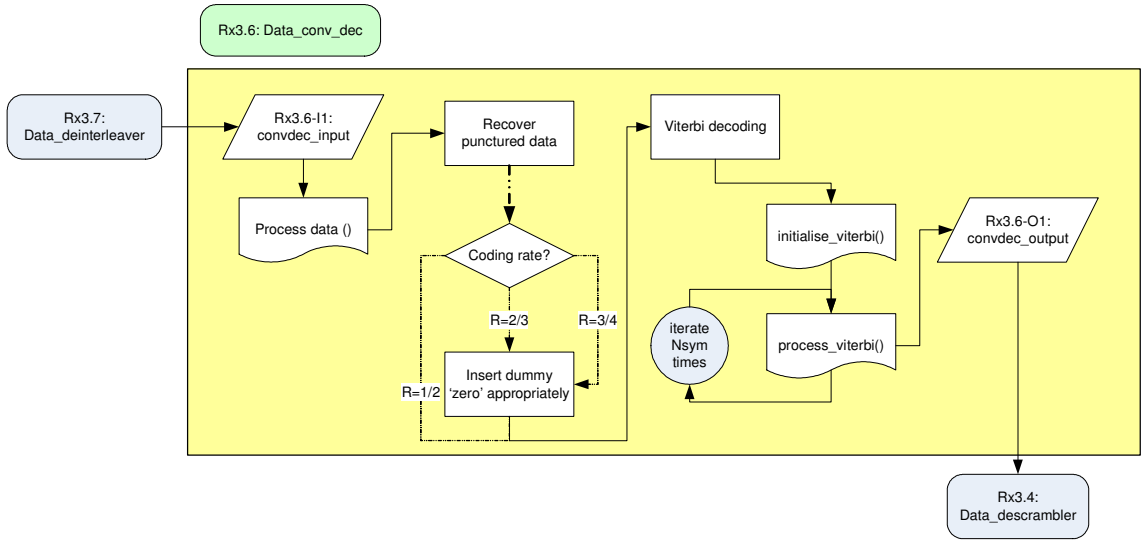

Figure 65. An example of Viterbi decoder: *DATA_conv_dec* functional flow.

Column	Description	Functions
1 … 6	Current_state	current 6 left most bits in the shift register
7 … 12	Next_state0	new 6 bits assume input '0'
13 … 18	Next_state1	new 6 bits assume input '1'
19, 20	Conv_output0	convolutional encoder output assume input '0'
21, 22	Conv_output1	convolutional encoder output assume input '1'

Table 18. Viterbi decoding lookup matrix.

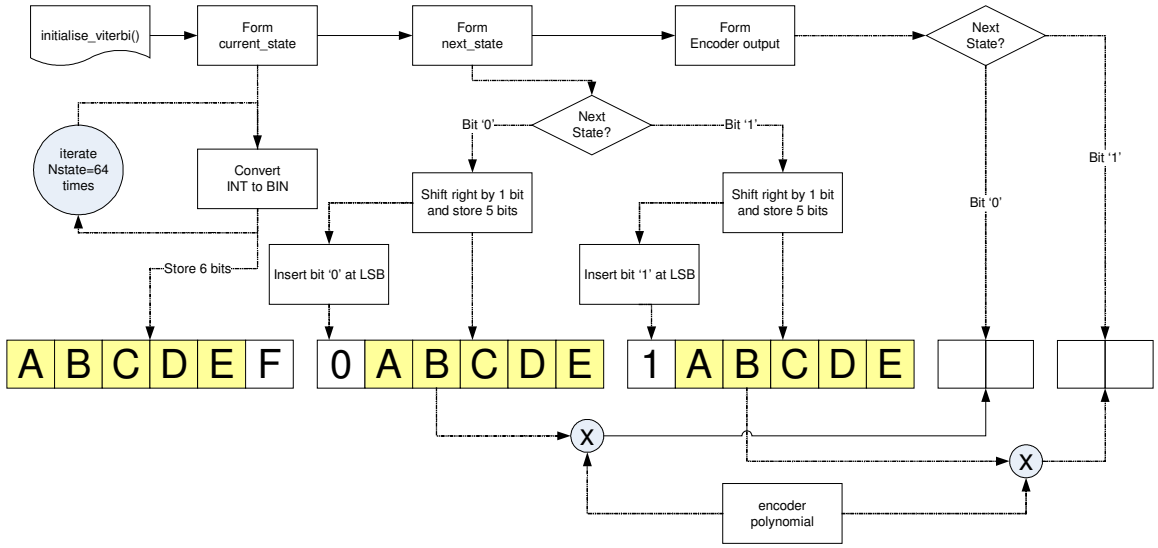

Figure 66. *initialise_viterbi()* functional flow.

b. Process_viterbi()

After the lookup matrix has been initialized, the overall data is divided into N_{SYM} fixed size bits streams for Viterbi decoding by the *process_viterbi()* function call. The *process_viterbi()* functional flow is shown in Figure 67. It consists of four main procedures: (1) select the corresponding insertion pattern for specific puncturing code, (2) initialize the relevant variables, (3) decode using the Viterbi algorithm and (4) make a decision by selecting the best code sequence. The code sequence is selected based on the path with the shortest Hamming distance. The actual Viterbi decoding is relatively complex and involved. It is carried out by the *BUTTERFLY_viterbi()* subroutine. Looking at Figure 67, we can deduce that *BUTTERFLY_viterbi()* will be called a total of $\left(N_{state} * N_{iter}\right)$ times, which could result in potentially lengthy code. Hence, a type-defined subroutine *BUTTERFLY_viterbi()* is defined in the header file: *Data_conv_dec.h* to reduce the length of the code.

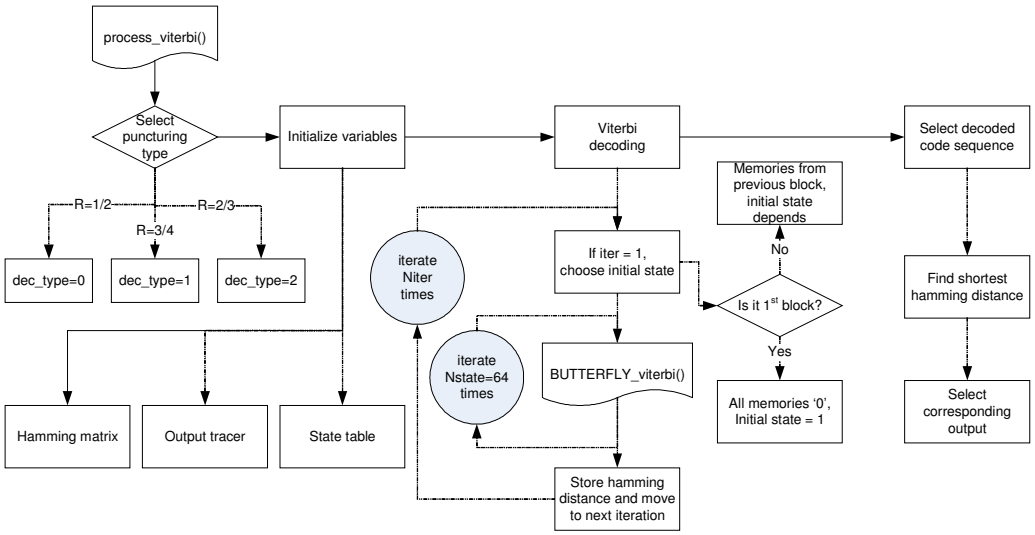

Figure 67. *process_viterbi()* functional flow.

c. BUTTERFLY_viterbi()

The *BUTTERFLY_viterbi()* functional flow is described in Figure 68. As shown, the expected next bit (either '0' or '1') determines the location on the lookup matrix that will be referenced. Dummy bits have been added to compensate for the puncturing done at the encoder for different code rate, R. These dummy bits shall NOT influence the Hamming distance for each iteration and, hence, will be ignored. A decision shall be made to choose the path with the shortest Hamming distance.

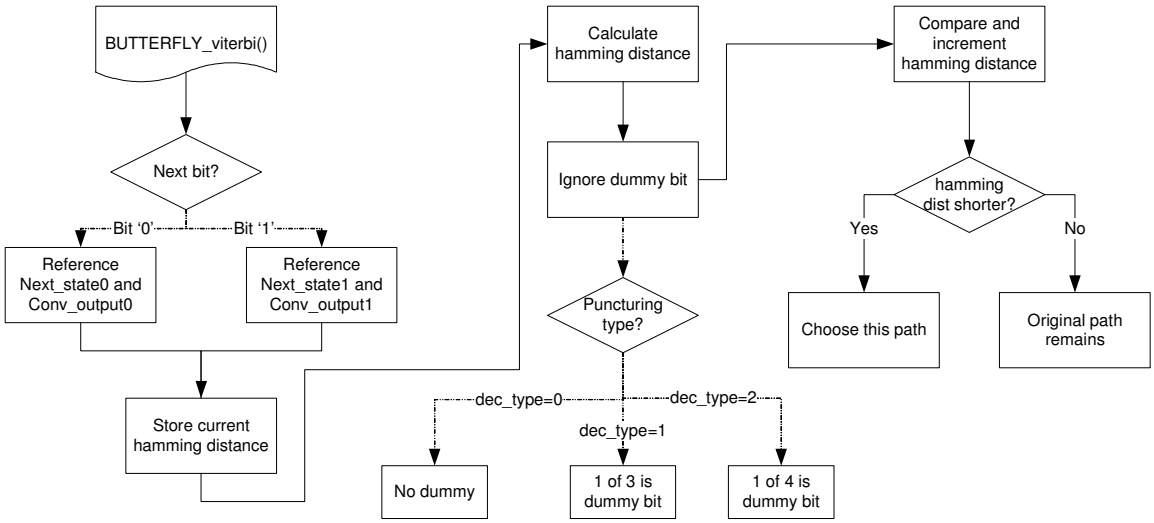

Figure 68. *BUTTERFLY_viterbi()* functional flow.

After $(N_{state} * N_{iter})$ iterations, the best path will be chosen based on the shortest Hamming distance. This is the output decoded stream that will be passed to *Data_descrambler* component as shown in Figure 65.

B. OTHER CHALLENGES

The challenges posted here are to raise awareness of potential considerations when coding using OSSIE. The newer version of OSSIE might have tackled the challenges but the following are with respect to the current version OSSIE 0.5.0.

1. Data Synchronisation (Ports Management)

Great care should be taken when passing parameters between components through the input and output ports. Handling of a thread between objects must be monitored closely so that there will not be a conflict in parameters and variables being called, which can lead to potential logic errors. This is especially critical in the IEEE 802.11a model as a component can be referenced a few times. For example, in the transmitter, *PPDU_map* is called three times to process preamble, SIGNAL and DATA subframes. There are common variables being used for different functional call, and the sequence of referencing different processes in the component is critical. The strategy here is to make use of control functions like *lock()* and *unlock()* in the defined object. Using *PPDU_map* as an example, the processes flow to maintain data integrity and prevent logic errors in the *process_data()* function call is shown in Figure 69.

2. MIMO Components

A typical component usually has a single input and single output (SISO) port configuration. However, as described above, IEEE 802.11a PHY model does consist of components with multiple inputs and multiple outputs (MIMO). One good example is *PPDU_map* in Figure 69. However, the OSSIE environment requires that each input (output) port must only be connected to another output (input) port. One must be careful to ensure same type of variables are passed between two ports of the same connection and be mindful of the potential logic errors described in Section B1. The table of SISO and MIMO components with their relevant port types is shown in APPENDIX A to demonstrate the potential confusion of different port types.

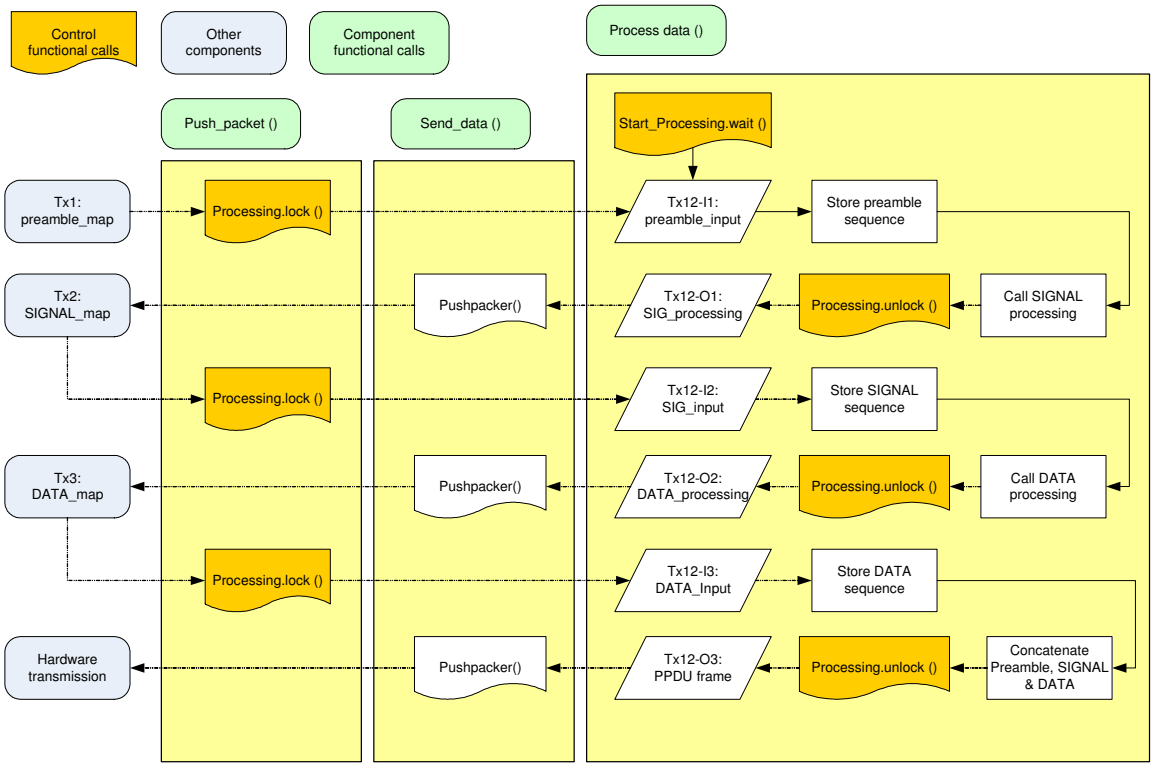

Figure 69. *PPDU_map* MIMO and control functional flow.

3. Control Variables

Control variables are passed between components for the normal functioning of the model. These parameters can either be global constant parameters or dynamic parameters that can be modified. For global constants (e.g. number of samples per FFT), the strategy is to define them in a header file (*global_para.h*) that can be assessed by all components. These constants are reproduced in APPENDIX B as a reference. As for dynamic parameters like RATE and LENGTH fields, these will be passed between components by prefixing them to the information transmitted as shown in Figure 70.

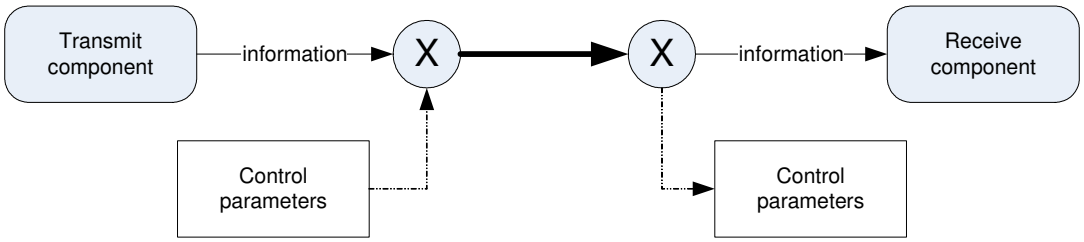

Figure 70. Transmission of dynamic control variables.

In this chapter, the major challenges that were encountered while developing the IEEE 802.11a PHY layer model were described. With the transceiver model developed, the next phase under the Incremental Development Model is to verify the functionalities of the various components. The next chapter shall focus on verifying these functionalities using test cases provided in the IEEE 802.11a standard.

VI. VERIFY

A. TRANSMITTER

In this section, functionalities of the transmitter components shall be verified based on test cases provided in Annex G of the IEEE 802.11a standard [3]. Similar to previous chapters, the section is broken down into preamble, SIGNAL and DATA subframes. The example in Annex G consists of ASCII information to be transmitted at a data rate of 36 Mbits/s and a total PSDU length of 100 octets (i.e. LENGTH = 100). The test results are divided into two categories: summarized and detailed traces. The summarized trace is attached in APPENDIX C, which verifies that all components carried out their functions accordingly. The entire transmitter test passes through 31 I/O sequential flows, and this is summarized in APPENDIX D. The detailed traces in this section are described according to (1) the functions of the component, (2) files where the test results are traced and stored, and (3) evaluation of the test results. All the files mentioned in this chapter have been included in the reference CD.

1. Preamble

Table 19 summarizes the components and their functions to form the preamble subframe. The test cases shall demonstrate these specific functions.

Index	Component	Functions
Tx1	**preamble_map**	- initiate the Tx routine - form short training (ST) and long training (LT) sequence - send preamble (ST + LT) to PPDU
Tx1.1	**short training (ST)**	
Tx1.1.9	**ST_carrier_map**	- ST carrier mapping
Tx1.1.10	**ST_IFFT**	- ST IFFT
Tx1.2	**long training (LT)**	
Tx1.2.9	**LT_carrier_map**	- LT carrier mapping
Tx1.2.10	**LT_IFFT**	- LT IFFT
Tx1.2.11	**LT_cyclicPrefix**	- LT cyclic prefix append

Table 19. IEEE 802.11a Transmitter preamble subframe components functionalities.

a. Tx1: Preamble Mapping (Assembly Controller)

This component carries out the following functions: (1) initiate the transmitter routine, (2) form ST and LT sequences, and (3) append the two sequences and form the preamble subframe. The detailed traces are captured in *preamble_map.txt* and a

summary of the traces are provided in Table 20. The traces show that 52 binary bits are sent out for processing to form 160 complex samples of ST sequence and follow by another set of 52 binary bits that are sent out to form 160 complex samples (I and Q channels) of LT sequence. The final sequence of 320 complex samples forms the preamble subframe.

No	Trace	Explanations
1	Processing short training sequence, size: 52	52 pre-defined I/Q real samples are sent out for processing to form the ST sequence.
2	Complex float pushpacket received, length 160	160 I/Q float samples of ST sequence received and stored for future transmission.
3	Processing long training sequence, size: 52	52 pre-defined I/Q real samples are sent out for processing to form the LT sequence
4	Complex float pushpacket received, length 160	160 I/Q float samples of LT sequence received and stored for future transmission
5	Processing preamble sequence, size: 320 Preamble_append Tx data, Length 320	ST and LT sequences are cascaded and sent to *PPDU_map* for future transmission

Table 20. *preamble_map* detail traces and explanations.

b. Tx1.1.9: Carriers Mapping (ST)

This component carries out the function of carriers mapping on the ST sequence. The detailed traces are captured in *st_carriers_map.txt* and a summary of the traces are provided in Table 21. The traces show that 52 binary bits are received and carriers-mapped to a size of 64 complex samples for IFFT.

No	Trace	Explanations
1	Complex short pushpacket received, length 52	52 pre-defined I/Q real samples received
2	Processing ST carrier mapping ST_carriers_map Tx data, Length 64	Mapped into 64 samples for IFFT

Table 21. *ST_carriers_map* detail traces and explanations.

c. Tx1.1.10: IFFT (ST)

This component carries out the function of IFFT on the ST sequence. The detailed traces are captured in *st_ifft.txt* and a summary of the traces are provided in Table 22. The traces show that 64 complex samples are received and gone through IFFT and duplication, to form 160 complex samples.

No	Trace	Explanations
1	Complex float pushpacket received, length 64	64 pre-defined I/Q float samples received
2	Processing ST IFFT ST_IFFT Tx data, Length 160	Mapped into 64 time samples after IFFT Duplicate to form 160 ST sequence

Table 22. *ST_IFFT* detail traces and explanations.

d. Tx1.2.9: Carriers Mapping (LT)

This component carries out the function of carriers mapping on the LT sequence. The detailed traces are captured in *lt_carriers_map.txt* and a summary of the traces are provided in Table 23. The traces show that 52 binary bits are received and carriers-mapped to a size of 64 complex samples for IFFT.

No	Trace	Explanations
1	Complex short pushpacket received, length 52	52 pre-defined I/Q real samples received
2	Processing LT carrier mapping LT_carriers_map Tx data, Length 64	Mapped into 64 samples for IFFT

Table 23. *LT_carriers_map* detail traces and explanations.

e. Tx1.2.10: IFFT (LT)

This component carries out the function of IFFT on the LT sequence. The detailed traces are captured in *lt_ifft.txt* and a summary of the traces are provided in Table 24. The traces show that 64 complex samples are received and gone through IFFT and duplication, to form 128 complex samples.

No	Trace	Explanations
1	Complex float pushpacket received, length 64	64 pre-defined I/Q float samples received
2	Processing LT IFFT LT_IFFT Tx data, Length 128	Mapped into 64 time samples after IFFT Duplicate to form 128 samples

Table 24. *LT_IFFT* detail traces and explanations.

f. Tx1.2.11: Cyclic Prefix (LT)

This component carries out the function of appending the cyclic prefix to the received complex samples. The detailed traces are captured in *lt_cyclicprefix.txt* and a summary of the traces are provided in Table 25. The traces show that 128 complex samples are received and cyclic prefix is appended to form the 160 complex samples of LT sequence.

No	Trace	Explanations
1	Complex float pushpacket received, length 128	128 I/Q time samples received
2	Processing cyclic prefix LT cyclic prefix modulated data, Length 160	Add cyclic prefix to form 160 LT sequence

Table 25. *LT_cyclicPrefix* detail traces and explanations.

2. SIGNAL

Table 26 summarizes the components and their functions to form the SIGNAL subframe. The test cases shall demonstrate these specific functions.

Index	Component	Functions
Tx2	**header_map (SIGNAL_map)**	- form SIGNAL (SIG) samples - send SIG to PPDU
Tx2.6	**SIG_conv_enc**	- SIG convolutional encoding
Tx2.7	**SIG_interleaver**	- SIG interleaving
Tx2.8	**SIG_BPSK_mod**	- SIG BPSK modulation
Tx2.9	**SIG_carriers_map**	- SIG carriers mapping
Tx2.10	**SIG_IFFT**	- SIG IFFT
Tx2.11	**SIG_cyclicprefix**	- SIG cyclic prefix

Table 26. IEEE 802.11a Transmitter SIGNAL subframe components functionalities.

a. Tx2: SIGNAL Mapping

This component forms the SIGNAL subframe and send the subframe to *PPDU_map*. The detailed traces are captured in *signal_map.txt* and a summary of the traces are provided in Table 27. The traces show that eight binary control bits are received to initiate the formation of the SIGNAL subframe. After the processing, the SIGNAL subframe, together with 16 control bits, are sent to *PPDU_map.*

No	Trace	Explanations
1	Real short pushpacket received, length 8	Received control bits from *PPDU_map* to start SIGNAL processing
2	Processing Header sequence, size: 24 Processing SIGNAL bits, size: 24	Retrieved RATE and LENGTH information. Form SIGNAL raw bits and send for processing
3	Complex float pushpacket received, length 80	80 I/Q float samples of SIGNAL subframe received and stored for future transmission
4	Processing SIGNAL sequence, size: 80 Header_append Tx data, Length 96	16 control bits of RATE and LENGTH appended to 80 I/Q time samples of SIGNAL subframe

Table 27. *SIGNAL_map* detail traces and explanations.

b. Tx2.6: Convolutional Encoder (SIG)

This component carries out the function of convolutional encoding for the SIGNAL field. The detailed traces are captured in *sig_conv_enc.txt* and a summary of the traces are provided in Table 28. The traces show that 24 binary bits are received and encoded to form 48 binary bits.

No	Trace	Explanations
1	Real short pushpacket received, length 24	24 SIGNAL raw bits received
2	Processing SIG convolution encoding, size: 24 SIG encoded Tx data, Length 48	Encoder R=1/2, hence output 48 encoded bits

Table 28. *SIG_conv_enc* detail traces and explanations.

c. Tx2.7: Interleaver (SIG)

This component carries out the function of interleaving the encoded SIGNAL field. The detailed traces are captured in *sig_interleaver.txt* and a summary of the traces are provided in Table 29. The traces show that 48 binary bits are received and interleaved to form 48 binary bits.

No	Trace	Explanations
1	Real short pushpacket received, length 48	48 SIGNAL encoded bits received
2	Processing SIG interleaver, size: 48	48 bits went through two permutations of interleaving

Table 29. *SIG_interleaver* detail traces and explanations.

d. Tx2.8: BPSK Modulation (SIG)

This component carries out the function of BPSK modulation on the interleaved bits. The detailed traces are captured in *sig_bpsk_mod.txt* and a summary of the traces are provided in Table 30. The traces show that 48 binary bits are received and modulated to a size of 48 complex BPSK samples.

No	Trace	Explanations
1	Real short pushpacket received, length 48	48 SIGNAL interleaved bits received
2	Processing BPSK modulation SIG modulated data, Length 48	Modulated into 48 BPSK I/Q float samples

Table 30. *SIG_BPSK_mod* detail traces and explanations.

e. Tx2.9: Carriers Mapping (SIG)

This component carries out the function of carriers mapping on the SIGNAL field. The detailed traces are captured in *sig_carriers_map.txt* and a summary of the traces are provided in Table 31. The traces show that 48 complex samples are received and carriers-mapped to a size of 64 complex samples for IFFT.

No	Trace	Explanations
1	Complex float pushpacket received, length 48	48 I/Q float samples received
2	Processing carrier mapping SIG carriers mapped data, Length 64	Mapped into 64 samples for IFFT

Table 31. *SIG_carriers_map* detail traces and explanations.

f. Tx2.10: IFFT (SIG)

This component carries out the function of IFFT on the SIGNAL field. The detailed traces are captured in *sig_ifft.txt* and a summary of the traces are provided in

Table 32. The traces show that 64 complex samples are received and gone through IFFT to form 64 complex samples.

No	Trace	Explanations
1	Complex float pushpacket received, length 64	64 I/Q float samples received
2	Processing SIG IFFT SIG_IFFT Tx data, Length 64	Mapped into 64 time samples after IFFT

Table 32. *SIG_IFFT* detail traces and explanations.

g. Tx2.11: Cyclic Prefix (SIG)

This component carries out the function of appending the cyclic prefix to the received complex samples. The detailed traces are captured in *sig_cyclicprefix.txt* and a summary of the traces are provided in Table 33. The traces show that 64 complex samples are received and cyclic prefix is appended to form the 80 complex samples of SIGNAL subframe.

No	Trace	Explanations
1	Complex float pushpacket received, length 64	64 I/Q time samples received
2	Processing SIG cyclic prefix SIG_cyclicPrefix Tx data, Length 80	Add cyclic prefix to form 80 SIGNAL subframe

Table 33. *SIG_cyclicPrefix* detail traces and explanations.

3. Data

Table 34 summarizes the components and their functions to form the DATA subframe. The test cases shall demonstrate these specific functions.

Index	Component	Functions
Tx3	**data_map**	- form time data samples from PSDU - send DATA samples to PPDU
Tx3.4	**data_scrambler**	- scramble the raw data
Tx3.5	**data_tail_replacement**	- replace tail with zeroes
Tx3.6	**data_conv_enc**	- data convolutional encoding - data puncturing
Tx3.7	**data_interleaver**	- data interleaving
Tx3.8	**data_mod_map**	- data modulation mapping
Tx3.9	**data_carriers_map**	- data carriers mapping
Tx3.10	**data_IFFT**	- data IFFT
Tx3.11	**data_cyclicprefix**	- data cyclic prefix
Tx0	**data_PSDU**	- input PSDU data

Table 34. IEEE 802.11a Transmitter DATA subframe components functionalities.

a. Tx3: DATA Mapping

This component forms the DATA subframe and send the subframe to *PPDU_map*. The detailed traces are captured in *data_map.txt* and a summary of the traces are provided in Table 35. The traces show that 24 binary control bits are received to initiate the formation of the DATA subframe. After retrieving the PSDU information bits, the binary bits are sent for processing to form the DATA subframe. The final step involves sending the 480 complex samples of DATA subframe to *PPDU_map*.

No	Trace	Explanations
1	Real short pushpacket received, length 24	Received control bits from *PPDU_map* to start DATA processing (include RATE and LENGTH)
2	Activate PSDU processing Data Tx Bits, Length 24	Activate transfer of PSDU to *DATA_map*
3	Real short pushpacket received, length 800	800 PSDU raw bits (100 octets) received
4	Send raw data for scrambling Data Tx Bits, Length 869	864 DATA bits send for processing (PAD bits added), 5 control bits (CBs) appended
5	Complex float pushpacket received, length 480	480 I/Q DATA time samples received
6	Send processed data to form PPDU Data_append Tx data, Length 480	Send DATA time samples to *PPDU_map*

Table 35. *DATA_map* detail traces and explanations.

b. Tx3.4: Scrambler (DATA)

This component carries out the function of scrambling the DATA field. The detailed traces are captured in *data_scrambler.txt* and a summary of the traces are provided in Table 36. The traces show that 864 binary bits are received and formed 864 scrambled bits. Note that there are five control bits being passed between components.

No	Trace	Explanations
1	Real short pushpacket received, length 869	864 (and 5 CBs) raw bits received
2	Processing data in the scrambler Scrambled data Bits, Length 869	864 scrambled bits sent out together with 5 CBs

Table 36. *DATA_scrambler* detail traces and explanations.

c. Tx3.5: Tail Replacement (DATA)

This component carries out the function of replacing the scrambled tail bits in the DATA field with non-scrambled “zero” bits. The detailed traces are captured in *data_tail_replace.txt* and a summary of the traces are provided in Table 37. The traces show that 864 scrambled bits are received and formed 864 bits after the tail bits are replaced. Note that there are five control bits being passed between components.

No	Trace	Explanations
1	Real short pushpacket received, length 869	864 (and 5 CBs) scrambled bits received
2	Processing tail replacement Tail replaced data Bits, Length 869	864 sent out together with 5 CBs after tail replacement

Table 37. *DATA_tail_replacement* detail traces and explanations.

d. Tx3.6: Convolutional Encoder (DATA)

This component carries out the function of convolutional encoding for the DATA field. The detailed traces are captured in *data_conv_enc.txt* and a summary of the traces are provided in Table 38. The traces show that 864 binary bits are received and encoded with a coding rate of $\frac{3}{4}$ (with puncturing) to form 1152 binary bits. Note that there are five control bits being passed between components.

No	Trace	Explanations
1	Real short pushpacket received, length 869	864 bits received
2	Processing Data convolution encoding, size: 864 Puncturing the encoded Data, size: 1152 Data encoded Tx data, Length 1157	36 Mbits/s : coding rate of 3/4, hence 1152 output bits from 864 input bits. 5 CBs added to the transmitted data

Table 38. *DATA_conv_enc* detail traces and explanations.

e. Tx3.7: Interleaver (DATA)

This component carries out the function of interleaving the encoded DATA field. The detailed traces are captured in *data_interleaver.txt* and a summary of the traces are provided in Table 39. The traces show that 1152 binary bits are received and interleaved to form 1152 binary bits. Note that there are five control bits being passed between components.

No	Trace	Explanations
1	Real short pushpacket received, length 1157	1152 DATA encoded bits received
2	Processing Data interleaver, size: 1152 Data interleaved Tx bits, Length 1157	1152 bits went through two permutations of interleaving

Table 39. *DATA_interleaver* detail traces and explanations.

f. Tx3.8: Modulation Mapping (DATA)

This component carries out the function of 16-QAM modulation on the interleaved bits. The detailed traces are captured in *data_mod_map.txt* and a summary of the traces are provided in Table 40. The traces show that 1152 binary bits are received

and modulated to a size of 288 complex 16-QAM samples. Note that there are five control bits being passed between components.

No	Trace	Explanations
1	Real short pushpacket received, length 1157	1152 DATA interleaved bits received
2	Processing 16QAM modulation Data modulated samples, Length 293	36 Mbits/s : 16-QAM modulation, hence modulated into 288 16-QAM I/Q float samples with 5 CBs

Table 40. *DATA_mod_map* detail traces and explanations.

g. Tx3.9: Carriers Mapping (DATA)

This component carries out the function of carriers mapping on the DATA field. The detailed traces are captured in *data_carriers_map.txt* and a summary of the traces are provided in Table 41. The traces show that 288 complex samples are received and carriers-mapped to a size of 384 complex samples for IFFT. Note that there is only one control bit being passed between components.

No	Trace	Explanations
1	Complex float pushpacket received, length 293	288 I/Q float samples received
2	Processing carrier mapping Data carriers mapped data, Length 385	Mapped into 6 x 64 samples for IFFT with 1 CB.

Table 41. *DATA_carriers_map* detail traces and explanations.

h. Tx3.10: IFFT (DATA)

This component carries out the function of IFFT on the DATA field. The detailed traces are captured in *data_ifft.txt* and a summary of the traces are provided in Table 42. The traces show that 384 complex samples are received and gone through IFFT to form 384 complex samples. Note that there is only one control bit being passed between components.

No	Trace	Explanations
1	Complex float pushpacket received, length 385	6 x 64 I/Q float samples received
2	Processing DATA IFFT DATA_IFFT Tx data, Length 385	Mapped into 64 time samples after IFFT (6 iterations), with 1 CB

Table 42. *DATA_IFFT* detail traces and explanations.

i. Tx3.11: Cyclic Prefix (DATA)

This component carries out the function of appending the cyclic prefix to the received complex samples. The detailed traces are captured in *data_cyclicprefix.txt* and a summary of the traces are provided in Table 43. The traces show that 384 complex samples are received and cyclic prefix is appended to form the 480 complex samples of DATA subframe.

No	Trace	Explanations
1	Complex float pushpacket received, length 385	6 x 64 I/Q time samples received
2	Processing Data cyclic prefix Data_cyclicPrefix Tx data, Length 480	Add cyclic prefix (6 x 16 samples) to form 480 DATA subframe

Table 43. *DATA_cyclicPrefix* detail traces and explanations.

4. PPDU (Final Concatenation)

The final piece to the transmitter is the function of concatenating the preamble, SIGNAL and DATA subframes together and form the PPDU frame. This main control is carried out by *PPDU_map* component.

a. Tx12: PPDU Mapping

This component forms the PPDU frame from the three subframes. The detailed traces are captured in *PPDU_map.txt* and a summary of the traces are provided in Table 44. The traces show that binary control bits are sent to initiate the formation of SIGNAL and DATA subframes. The traces also show the retrieval of the preamble, SIGNAL and DATA subframes. The final PPDU frame consists of 880 complex samples.

No	Trace	Explanations
1	Complex float pushpacket received, length 320	480 I/Q Preamble time samples received
2	Sent Header control bits CB Tx data, Length 8	Activate SIGNAL subframe processing
3	Complex float pushpacket received, length 96	80 I/Q SIGNAL time samples received, with 16 CBs (RATE and LENGTH)
4	Sent DATA control bits CB Tx data, Length 24	Activate DATA subframe processing
5	Complex float pushpacket received, length 480	480 I/Q DATA time samples received
6	Processing PPDU sequence, size: 880	Final PPDU frame of 880 I/Q time samples to be sent out for transmission

Table 44. *PPDU_map* detail traces and explanations

B. RECEIVER

In this section, functionalities of the receiver components shall be verified. As annex G in the standard only described the transmitter outputs, receiver components are tested on their abilities to regenerate the transmitter input data (e.g. the receiver decoder outputs should be similar to the transmitter encoder input). Similar to the previous section, this section is broken down into preamble, SIGNAL and DATA subframes, and the test results are divided into two categories: summarized and detailed traces. The summarized trace is attached in APPENDIX C. The entire receiver test passes through 20 I/O sequential flows, and this is summarized in APPENDIX D. The detailed traces in this section are described according to (1) the functions of the component, (2) files where the test results are traced and stored, and (3) evaluation of the test results. All the files mentioned in this chapter have been included in the reference CD.

1. Preamble

Table 45 summarizes the components and their functions to remove the preamble subframe. The test cases shall demonstrate these specific functions.

Index	Component	Functions
Rx0	**Rx_data**	- received digitized data stream
Rx1 / Rx12	**PPDU_rx**	- extract the required digitized PPDU stream - removed preamble from PPDU - send stream for header removal

Table 45. IEEE 802.11a Receiver preamble subframe components functionalities.

a. Rx0: Receiver Data (Assembly Controller)

This component simulates the retrieval of digitize data stream. The detailed traces are captured in *rx_data.txt* and a summary of the traces are provided in Table 46. The traces show that 884 complex samples are received.

No	Trace	Explanations
1	Start PPDU digitized receiver stream_sizeI = 884, stream_sizeQ = 884	880 time samples received. 4 additional arbitrary 'noise' bits prefix to the stream

Table 46. *Rx_data* detail traces and explanations.

b. Rx1: PPDU Receiver

This component carries out the following functions: (1) extract the digitized PPDU stream, (2) remove preamble from PPDU, and (3) send stream for SIGNAL subframe removal. The detailed traces are captured in *ppdu_rx.txt* and a

summary of the traces are provided in Table 47. The traces show that 320 complex samples belonging to the preamble subframe are extracted from the received samples. The remaining 560 complex samples are sent out for processing.

No	Trace	Explanations
1	Removed preamble from PPDU samples preamble_sizeI = 320, preamble_sizeQ = 320	Proceed to remove the 320 time samples of Preamble subframe
2	Sent PPDU (preamble removed) samples PPDU_preamble_removed data, Length 560	560 time samples (SIGNAL + DATA subframes) ready to be forwarded

Table 47. *PPDU_rx* detail traces and explanations.

2. SIGNAL

Table 48 summarizes the components and their functions to extract RATE and LENGTH from SIGNAL subframe. The test cases shall demonstrate these specific functions.

Index	Component	Functions
Rx2	**Header_rx (SIGNAL_rx)**	- remove SIG from PPDU - send header for processing - extract RATE & LENGTH from SIG - send received data for processing
Rx2.11	**SIG_cyclicprefix_rem**	- SIG cyclic prefix removal
Rx2.10	**SIG_FFT**	- SIG FFT
Rx2.9	**SIG_carriers_demap**	- SIG carriers demapping
Rx2.8	**SIG_BPSK_demod**	- SIG BPSK demodulation
Rx2.7	**SIG_deinterleaver**	- SIG deinterleaving
Rx2.6	**SIG_conv_dec**	- SIG convolutional decoding

Table 48. IEEE 802.11a Receiver SIGNAL subframe components functionalities.

a. Rx2: SIGNAL Receiver

This component carries out the following functions: (1) remove SIGNAL from PPDU, (2) send SIGNAL subframe to retrieve RATE and LENGTH, and (3) send DATA subframe for PSDU retrieval. The detailed traces are captured in *header_rx.txt* and a summary of the traces are provided in Table 49. The traces show that 560 complex samples are received and 80 complex samples belonging to SIGNAL subframe are sent for processing to retrieve the RATE and LENGTH information bits. The remaining 480 complex samples belonging to DATA subframe are sent out together with five control bits.

No	Trace	Explanations
1	Complex float pushpacket received, length 560	560 I/Q samples (SIGNAL + DATA subframes) received
2	Sent SIG time samples for processing ... Header removed data output, Length 80	80 I/Q samples of SIGNAL subframe sent for processing
3	Real short pushpacket received, length 24 Processing Data sequence, rate: 36 Mbits/s, length: 100 octets	24 single stream of SIGNAL bits received, RATE= 36 Mbits/s; LENGTH=100 extracted
4	Send raw data for decoding Header removed data output, Length 485	480 DATA I/Q time samples sent for processing, with 5 CBs

Table 49. *SIGNAL_rx* detail traces and explanations.

b. Rx2.11: Cyclic Prefix Removal (SIG)

This component carries out the function of removing the cyclic prefix from the SIGNAL subframe. The detailed traces are captured in *sig_cyclicprefix_rem.txt* and a summary of the traces are provided in Table 50. The traces show that 80 complex samples are received and cyclic prefix is removed to form the 64 complex samples.

No	Trace	Explanations
1	Complex float pushpacket received, length 80	80 I/Q SIGNAL time samples received
2	Processing SIG cyclic prefix remove SIG_cyclicPrefix_rem Tx data, Length 64	64 I/Q time samples extracted for FFT

Table 50. *SIG_cyclicPrefix_rem* detail traces and explanations.

c. Rx2.10: FFT (SIG)

This component carries out the function of FFT on the SIGNAL subframe. The detailed traces are captured in *sig_fft.txt* and a summary of the traces are provided in Table 51. The traces show that 64 complex samples are received and gone through FFT to form 64 complex samples.

No	Trace	Explanations
1	Complex float pushpacket received, length 64	64 I/Q float samples received
2	Processing SIG FFT SIG_FFT Tx data, Length 64	Mapped into 64 complex subcarriers after FFT

Table 51. *SIG_FFT* detail traces and explanations.

d. Rx2.9: Carriers Demapper (SIG)

This component carries out the function of carriers demapping on the SIGNAL subframe. The detailed traces are captured in *sig_carriers_demap.txt* and a

summary of the traces are provided in Table 52. The traces show that 64 complex samples are received and carriers-demapped to a size of 48 complex samples.

No	Trace	Explanations
1	Complex float pushpacket received, length 64	64 I/Q subcarriers received
2	Processing carrier demapping SIG carriers demapped data, Length 48	Mapped into 48 complex constellations for BPSK demodulation

Table 52. *SIG_carriers_demap* detail traces and explanations.

e. Rx2.8: BPSK Demodulator (SIG)

This component carries out the function of BPSK demodulation on the complex samples. The detailed traces are captured in *sig_bpsk_demod.txt* and a summary of the traces are provided in Table 53. The traces show that 48 complex samples are received and demodulated to a size of 48 binary bits.

No	Trace	Explanations
1	Complex float pushpacket received, length 48	48 BPSK I/Q float samples received
2	Processing SIG BPSK demodulation SIG demodulated data, Length 48	Demodulated into 48 serial bits

Table 53. *SIG_BPSK_demod* detail traces and explanations.

f. Rx2.7: De-Interleaver (SIG)

This component carries out the function of deinterleaving the demodulated SIGNAL field. The detailed traces are captured in *sig_deinterleaver.txt* and a summary of the traces are provided in Table 54. The traces show that 48 binary bits are received and deinterleaved to form 48 binary bits.

No	Trace	Explanations
1	Real short pushpacket received, length 48	48 SIGNAL demodulated bits received
2	Processing SIG deinterleaver, size: 48 SIG deinterleaved Tx data, Length 48	48 bits went through two permutations of deinterleaving

Table 54. *SIG_deinterleaver* detail traces and explanations.

g. Rx2.6: Convolutional Decoder (SIG)

This component carries out the function of convolutional decoding on the SIGNAL field. The detailed traces are captured in *sig_conv_dec.txt* and a summary of the traces are provided in Table 55. The traces show that 48 binary bits are received and decoded to form 24 binary bits.

No	Trace	Explanations
1	Real short pushpacket received, length 48	48 SIGNAL bits received for decoding
2	Processing SIG convolution decoding, size: 48 return from initialise viterbi return from process viterbi SIG decoded Tx data, Length 24	Show bits went through initialize_viterbi() and process_viterbi(). 24 bits produced at end of decoding.

Table 55. *SIG_conv_dec* detail traces and explanations.

3. Data

Table 56 summarizes the components and their functions to retrieve PSDU from the DATA subframe. The test cases shall demonstrate these specific functions.

Index	Component	Functions
Rx3	**data_rx**	- receive and send raw data for processing - receive and send PSDU data to MAC layer
Rx3.11	**data_cyclicprefix_rem**	- data cyclic prefix removal
Rx3.10	**data_FFT**	- data FFT
Rx3.9	**data_carriers_demap**	- data carriers demapping
Rx3.8	**data_demod_map**	- data demodulation mapping
Rx3.7	**data_deinterleaver**	- data deinterleaving
Rx3.6	**data_conv_dec**	- data dummy insertion - data convolutional decoding
Rx3.5	**data_tail_replace**	- not required, encompass in descrambler
Rx3.4	**data_descrambler**	- descramble the raw data

Table 56. IEEE 802.11a Receiver DATA subframe components functionalities.

a. Rx3: DATA Receiver

This component carries out the functions of sending the DATA subframe for processing and retrieving the PSDU information bits. The detailed traces are captured in *data_rx.txt* and a summary of the traces are provided in Table 57. The traces show that 480 complex samples are sent for processing and 864 binary bits belonging to DATA field are received. The 800 PSDU information bits are sent out together with a control bit.

No	Trace	Explanations
1	Complex float pushpacket received, length 485	480 I/Q samples (DATA subframes) received, with 5 CBs
2	Sent raw data for processing ... Data output, Length 485	480 I/Q samples of DATA subframe sent for processing, with 5 CBs
3	Real short pushpacket received, length 864	864 DATA bits received (including PAD bits)
4	Send PSDU data to MAC layer ... PSDU Data output, Length 801	800 PSDU bits sent out, including 1 bit of LENGTH field

Table 57. *DATA_rx* detail traces and explanations.

b. Rx3.11: Cyclic Prefix Removal (DATA)

This component carries out the function of removing the cyclic prefix from the DATA subframe. The detailed traces are captured in *data_cyclicprefix_rem.txt* and a summary of the traces are provided in Table 58. The traces show that 480 complex samples are received and cyclic prefix is removed to form the 384 complex samples. Note that there are five control bits being passed between components.

No	Trace	Explanations
1	Complex float pushpacket received, length 485	480 I/Q DATA time samples received, with 5 CBs
2	Processing Data cyclic prefix remove Data_cyclicPrefix_rem Tx data, Length 389	6 x 64 I/Q time samples extracted for FFT, with 5 CBs

Table 58. *DATA_cyclicPrefix_rem* detail traces and explanations.

c. Rx3.10: FFT (DATA)

This component carries out the function of FFT on the DATA subframe. The detailed traces are captured in *data_fft.txt* and a summary of the traces are provided in Table 59. The traces show that 384 complex samples are received and gone through FFT to form 384 complex samples. Note that there are five control bits being passed between components.

No	Trace	Explanations
1	Complex float pushpacket received, length 389	6 x 64 I/Q float samples received, with 5 CBs
2	Processing DATA FFT DATA_FFT Tx data, Length 389	Mapped into 6 x 64 complex subcarriers after FFT, with 5 CBs

Table 59. *DATA_FFT* detail traces and explanations.

d. Rx3.9: Carriers Demapper (DATA)

This component carries out the function of carriers demapping on the DATA subframe. The detailed traces are captured in *data_carriers_demap.txt* and a summary of the traces are provided in Table 60. The traces show that 384 complex samples are received and carriers-demapped to a size of 288 complex samples. Note that there are five control bits being passed between components.

No	Trace	Explanations
1	Complex float pushpacket received, length 389	6 x 64 I/Q subcarriers received, with 5 CBs
2	Processing data carrier demapping Data carriers demapped data, Length 293	Mapped into 6 x 48 complex constellations for demodulation, with 5 CBs

Table 60. *DATA_carriers_demap* detail traces and explanations.

e. Rx3.8: Demodulation Mapper (DATA)

This component carries out the function of 16-QAM demodulation on the complex samples. The detailed traces are captured in *data_demod_map.txt* and a summary of the traces are provided in Table 61. The traces show that 288 complex samples are received and demodulated to a size of 1152 binary bits. Note that there are five control bits being passed between components.

No	Trace	Explanations
1	Complex float pushpacket received, length 293	6 x 48 I/Q float samples received, with 5 CBs
2	Processing Data demodulation Conduct 16QAM demodulation Data demodulated data, Length 1157	16-QAM demodulation carried out Demodulated into (288 x 2 x 2) 1152 serial bits, with 5 CBs

Table 61. *DATA_demod_map* detail traces and explanations.

f. Rx3.7: De-Interleaver (DATA)

This component carries out the function of deinterleaving the demodulated DATA field. The detailed traces are captured in *data_deinterleaver.txt* and a summary of the traces are provided in Table 62. The traces show that 1152 binary bits are received and deinterleaved to form 1152 binary bits. Note that there are five control bits being passed between components.

No	Trace	Explanations
1	Real short pushpacket received, length 1157	1152 DATA demodulated bits received
2	Processing Data deinterleaver Data deinterleaved Tx data, Length 1157	1152 bits went through two permutations of deinterleaving, excluding 5 CBs

Table 62. *DATA_deinterleaver* detail traces and explanations.

g. Rx3.6: Convolutional Decoder (DATA)

This component carries out the function of convolutional decoding on the DATA field. The detailed traces are captured in *data_conv_dec.txt* and a summary of the traces are provided in Table 63. The traces show that 1152 binary bits are received and decoded to form 864 binary bits. Note that there are five control bits being passed between components.

No	Trace	Explanations
1	Real short pushpacket received, length 1157	1152 DATA bits received
2	Processing DATA convolution decoding return from initialise viterbi return from process viterbi DATA decoded Tx data, Length 869	Show bits went through initialize_viterbi() and process_viterbi(). 864 bits obtained at end of dummy insertion and decoding, with 5 CBs

Table 63. *DATA_conv_dec* detail traces and explanations.

h. Rx3.4: Descrambler (DATA)

This component carries out the function of descrambling the DATA bits. The detailed traces are captured in *data_descrambler.txt* and a summary of the traces are provided in Table 64. The traces show that 869 binary bits are received and formed 864 descrambled bits.

No	Trace	Explanations
1	Real short pushpacket received, length 869	864 DATA bits received after decoding
2	Processing data in the descrambler Descrambled data Bits, Length 864	864 DATA bits descrambled (including PAD bits)

Table 64. *DATA_descrambler* detail traces and explanations.

In this chapter, the functionalities of the transmitter and receiver OSSIE models have been verified using test cases provided in the IEEE 802.11a standard. With this success in mind, the next chapter shall conclude the thesis research and provide recommendations for further research.

VII. CONCLUSION

A. SUMMARY

Software defined radio has been the emerging trend of radio design both in the commercial and military arena. Wireless LAN standards like IEEE 802.11a have been among the popular physical means of data transmission. This thesis lays the groundwork for implementing an IEEE 802.11a standard using open source software for SDR design. Critical functionalities at the Physical layer have been implemented and the convenience and flexibilities of using software to implement a popular radio standard as compared to expensive and rigid radio implementation using hardware components demonstrated.

In this thesis, we have successfully met the objectives defined in Chapter I:

1. The IEEE 802.11a PHY layer transmitter has been built using a total of 23 OSSIE components with 12 different functionalities and 31 sequential I/O processes. Correspondingly, the receiver is implemented using 18 components with 12 different functionalities and 20 sequential I/O processes.

2. All these components have been designed with modularity and flexibility in mind so that they contribute to the pool of components for future radio design. Most of the functionalities reside in the *process_data()* functional call within the component C++ file for standardization and ease of debugging. "*Readme*" files are also included in each component's directory to explain its I/O data types, functionalities and assumptions. Appropriate parameters can be modified easily for use in other transceivers. All the files mentioned in this research have been included in the reference CD.

3. With the design implemented fully in OWD environment, the SDR conforms to Software Communications Architecture (SCA) and the Common Object Request Broker Architecture (CORBA). This will ensure flexibility, performance and maximum potential for software module reuse.

4. Using the test cases provided in Annex G of the IEEE 802.11a standard document, all the components have been verified to provide the necessary functionalities expected of them.

OSSIE, being developmental software, has yet to release its full version. Most of the efforts from the OSSIE developers are channeled to fix bugs and enhance the software, rather then using the software to develop communications standards. This thesis leverages on the capabilities of the software, adapts it to a popular communication standard and advances OSSIE capabilities by demonstrating that such a marriage can be implemented with an integration of OSSIE components into a working waveform.

The Incremental Development Model was chosen for this thesis, which is comprised of three stages: Design, Develop and Verify. The advantage of this model is its incremental nature, which allows the developer to learn from earlier versions of the system and enhance the subsequent design. It provides a systematic approach of meeting the objectives of the thesis by adding verified components into the library and eventually forming the final product.

B. RECOMMENDATIONS

The software components developed here shall serve as a baseline to link up with other software or hardware components to implement a fully functional IEEE 802.11a transceiver. For this to happen, the following are potential areas to address in order to implement such a functional transceiver:

a) Map up the hardware resources needed such as the type of General Purpose Processor (GPP) or Field Programmable Gate Array (FPGA) to implement the functions of the components. The Universal Software Radio Peripheral (USRP) board is a hardware option to be considered as part of the RF front end and is a low cost and high-speed hardware component suitable to implement research-based software radio applications. A good reference is a course project setup by Virginia Tech that utilizes USRP and OSSIE to implement a SDR receiver [8].

b) Synchronization of the receiver for packets detection. As this research is done at baseband, it will be interesting to observe its performance in the 5GHz carrier frequencies range for the IEEE 802.11a standard. These filtering and synchronizing functions at higher frequencies would most likely be

implemented using hardware, but the possibilities of extending the software capabilities to the RF front end should also be considered.

c) Presence of channel noise and multi-path fading may lead to amplitude and phase errors in the received signals. The model needs to be modified to compensate for such perturbations. One proposal is to consider the use of diversity SDR receiver with central combiner

With the potential of implementing a fully functional radio standard, the follow up could be to use the developed components to test out the channel performances like Bit Error Rates (BER). Since the SDR is supposed to be modular and reconfigurable, its ability to be flexible in a dynamically changing environment can be further explored by changing parameters like the information bit rates in real time.

Academically, collaboration and research with Virginia Tech can be enhanced with this family of components. The experiences and development carried out in this thesis can also be exemplified for SDR education and training.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: IEEE 802.11A COMPONENTS PORT TYPES

A. TRANSMITTER

Index	Component	I/O type	Port name	Port type	Port name	Port type	Port name	Port type	Port name	Port type	Port name	Port type	Port name	Port type
Preamble subframe														
Tx1	**preamble_map**	MIMO	Tx1-O1	CS	Tx1-I1	CF	Tx1-O2	CS	Tx1-I2	CF	Tx1-O3	CF		
Tx1.1	**short training (ST)**													
Tx1.1.9	**ST_carrier_map**	SISO	Tx1.1.9-I1	CS	Tx1.1.9-O1	CF								
Tx1.1.10	**ST_IFFT**	SISO	Tx1.1.10-I1	CF	Tx1.1.10-O1	CF								
Tx1.2	**long training (LT)**													
Tx1.2.9	**LT_carrier_map**	SISO	Tx1.2.9-I1	CS	Tx1.2.9-O1	CF								
Tx1.2.10	**LT_IFFT**	SISO	Tx1.2.10-I1	CF	Tx1.2.10-O1	CF								
Tx1.2.11	**LT_cyclicPrefix**	SISO	Tx1.2.11-I1	CF	Tx1.2.11-O1	CF								
SIGNAL subframe														
Tx2	**header_map**	MIMO	Tx2-I1	RS	Tx2-O1	RS	Tx2-I2	CF	Tx2-O2	CF				
Tx2.6	**SIG_conv_enc**	SISO	Tx2.6-I1	RS	Tx2.6-O1	RS								
Tx2.7	**SIG_interleaver**	SISO	Tx2.7-I1	RS	Tx2.7-O1	RS								
Tx2.8	**SIG_BPSK_mod**	SISO	Tx2.8-I1	RS	Tx2.8-O1	CF								
Tx2.9	**SIG_carriers_map**	SISO	Tx2.9-I1	CF	Tx2.9-O1	CF								
Tx2.10	**SIG_IFFT**	SISO	Tx2.10-I1	CF	Tx2.10-O1	CF								
Tx2.11	**SIG_cyclicprefix**	SISO	Tx2.11-I1	CF	Tx2.11-O1	CF								
DATA subframe														
Tx3	**data_map**	MIMO	Tx3-I1	CF	Tx3-O1	RS	Tx3-I2	RS	Tx3-O2	RS	Tx3-I3	CF	Tx3-O3	CF
Tx3.4	**data_scrambler**	SISO	Tx3.4-I1	RS	Tx3.4-O1	RS								
Tx3.5	**data_tail_replacement**	SISO	Tx3.5-I1	RS	Tx3.5-O1	RS								
Tx3.6	**data_conv_enc**	SISO	Tx3.6-I1	RS	Tx3.6-O1	RS								
Tx3.7	**data_interleaver**	SISO	Tx3.7-I1	RS	Tx3.7-O1	RS								
Tx3.8	**data_mod_map**	SISO	Tx3.8-I1	RS	Tx3.8-O1	CF								
Tx3.9	**data_carriers_map**	SISO	Tx3.9-I1	CF	Tx3.9-O1	CF								
Tx3.10	**data_IFFT**	SISO	Tx3.10-I1	CF	Tx3.10-O1	CF								
Tx3.11	**data_cyclicprefix**	SISO	Tx3.11-I1	CF	Tx3.11-O1	CF								
Tx12	**PPDU_map**	MIMO	Tx12-I1	CF	Tx12-O1	RS	Tx12-I2	CF	Tx12-O2	CF	Tx12-I3	CF	Tx12-O3	CF
Tx0	**data_PSDU**	SISO	Tx0-I1	RS	Tx0-O1	RS								

Legend: **RS (real short): single stream, integer**
CS (complex short): dual I/Q channels, integer
CF (complex float): dual I/Q channels, float

B. RECEIVER

Index	Component	I/O type	Port name	Port type	Port name	Port type	Port name	Port type	Port name	Port type
Preamble subframe										
Rx0	**Rx_data**	SISO	Rx0-I1	CF	Rx0-O1	CF				
Rx1 / Rx12	**PPDU_rx**	SISO	Rx1-I1	CF	Rx1-O1	CF				
SIGNAL subframe										
Rx2	**Header_rx**	MIMO	Rx2-I1	CF	Rx2-O1	CF	Rx2-I2	RS	Rx2-O2	CF
Rx2.11	**SIG_cyclicprefix_rem**	SISO	Rx2.11-I1	CF	Rx2.11-O1	CF				
Rx2.10	**SIG_FFT**	SISO	Rx2.10-I1	CF	Rx2.10-O1	CF				
Rx2.9	**SIG_carriers_demap**	SISO	Rx2.9-I1	CF	Rx2.9-O1	CF				
Rx2.8	**SIG_BPSK_demod**	SISO	Rx2.8-I1	CF	Rx2.8-O1	RS				
Rx2.7	**SIG_deinterleaver**	SISO	Rx2.7-I1	RS	Rx2.7-O1	RS				
Rx2.6	**SIG_conv_dec**	SISO	Rx2.6-I1	RS	Rx2.6-O1	RS				
DATA subframe										
Rx3	**data_rx**	MIMO	Rx3-I1	CF	Rx3-O1	CF	Rx3-I2	RS	Rx3-O2	RS
Rx3.11	**data_cyclicprefix_rem**	SISO	Rx3.11-I1	CF	Rx3.11-O1	CF				
Rx3.10	**data_FFT**	SISO	Rx3.10-I1	CF	Rx3.10-O1	CF				
Rx3.9	**data_carriers_demap**	SISO	Rx3.9-I1	CF	Rx3.9-O1	CF				
Rx3.8	**data_demod_map**	SISO	Rx3.8-I1	CF	Rx3.8-O1	RS				
Rx3.7	**data_deinterleaver**	SISO	Rx3.7-I1	RS	Rx3.7-O1	RS				
Rx3.6	**data_conv_dec**	SISO	Rx3.6-I1	RS	Rx3.6-O1	RS				
Rx3.5	**data_tail_replace**	SISO	Rx3.5-I1	RS	Rx3.5-O1	RS				
Rx3.4	**data_descrambler**	SISO	Rx3.4-I1	RS	Rx3.4-O1	RS				
Rx3.0	**data_PSDU**	SISO	Rx3.0-I1	RS	Rx0-O1	RS				

Legend: **RS (real short): single stream, integer**
CF (complex float): dual I/Q channels, float

APPENDIX B: GLOBAL PARAMETERS

```
/*************************************************************************

This file lists all the global constants referenced in the OSSIE
IEEE 802.11a Transceiver Design.

*************************************************************************/

/* Transmitter control parameters */
const float Fsym = 20.0;              // OFDM symbol frequency spacing (20 MHz)
const float Fsub = Fsym / 64.0;       // OFDM symbol subcarrier freq spacing (20MHz/64)
const float Tifft = 1.0 / Fsub;       // IFFT / FFT period (3.2 micro-second)
const float Tshort = 10.0*Tifft/4.0; // short training sequence duration (8 micro-second)
const float Tgi2 = Tifft/2.0;         // training symbol GI duration (1.6 micro-second)
const float Tlong = Tgi2 + 2.0*Tifft; // long training sequence duration (8 micro-second)
const float Tpreamble = Tshort + Tlong; // PLCP preamble duration (16 micro-second)
const float Tcp = Tifft/4.0;          // cyclic prefix for header (0.8 micro-second)
const float Theader = Tifft+ Tcp;     // PLCP header duration (4 micro-second)
const float Tsample = 1.0 / Fsym;     // sample period (0.05 micro-second)
const float null=0.0;

const int octet=8;                    // size of an octet
const int Nrb = 4;                    // no. of RATE bits
const int Nlb = 12;                   // no. of LENGTH bits
const int Ntb = 6;                    // no. of TAIL bits
const int Nsb = 24;                   // no. of SIGNAL bits
const int Nserb = 16;                 // no. of SERVICE bits
const int Nscrb = 7;                  // no. of syn descramble bits
const int RES=0;                      // reserve for future use
const int Nsd = 48;                   // no. of data subcarriers
const int Nsp = 4;                    // no. of pilot subcarriers
const int Nst = Nsd + Nsp;            // no. of total subcarriers
const int Ntr = 52;                   // no. of bits per training symbol
const int guard_len = 12;             // total guard band carriers
const int ifft_len = Nst + guard_len; // total OFDM IFFT length
const int num_samples = ifft_len;     // no. of sub_carriers
const int cb_length = 8;              // length of control bits
const int cp_len_DATA = ifft_len/4;   // DATA and SIG cyclic prefix length
const int cp_len_preamble = ifft_len/2; // Preamble cyclic prefix length

/* Receiver control parameters */
const float Fsym = 20.0;              // OFDM symbol frequency spacing (20 MHz)
const float Fsub = Fsym / 64.0;       // OFDM symbol subcarrier freq spacing (20MHz/64)
const float Tifft = 1.0 / Fsub;       // IFFT / FFT period (3.2 micro-second)
const float Tcp = Tifft/4.0;          // cyclic prefix (0.8 micro-second)
const float Theader = Tifft+ Tcp;     // PLCP header duration (4 micro-second)
const float Tsample = 1.0 / Fsym;     // sample period (0.05 micro-second)
const float null = 0.0;

const int octet=8;
const int Nrb = 4;                    // no. of RATE bits
const int Nlb = 12;                   // no. of LENGTH bits
const int Ntb = 6;                    // no. of TAIL bits
const int Nsb = 24;                   // no. of SIGNAL bits
const int Nserb = 16;                 // no. of SERVICE bits
const int Nscrb = 7;                  // no. of syn descramble bits (7 bits)
const int Nsd = 48;                   // no. of data subcarriers
const int Nsp = 4;                    // no. of pilot subcarriers
const int Nst = Nsd + Nsp;            // no. of total subcarriers
const int guard_len = 12;             // total guard band carriers
const int fft_len = Nst + guard_len;  // total OFDM IFFT length
const int cp_len = fft_len/4;         // cyclic prefix length
const int num_samples = fft_len;      // no. of sub_carriers
const int Nsr=6;                      // no. of shift registers
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: SUMMARIZED TRACE

A. TRANSMITTER

```
************************************
Welcome to OSSIE simulation: 802.11a PPDU transmission
************************************
Start process data function
Processing short training sequence, size: 52
Tx training data, Length 52
************************************
Start ST_carriers_map function
************************************
Complex short pushpacket received, length 52
Processing ST carrier mapping
ST_carriers_map Tx data, Length 64
************************************
Start ST_IFFT function
************************************
Complex float pushpacket received, length 64
Processing ST IFFT
IFFT time output (Magnitude), Length 64
ST_IFFT Tx data, Length 160
************************************
Start Preamble_append function
************************************
Complex float pushpacket received, length 160
Processing short training sequence storage, size: 160
Processing long training sequence, size: 52
Tx training data, Length 52
************************************
Start LT_carriers_map function
************************************
Complex short pushpacket received, length 52
Processing LT carrier mapping
LT_carriers_map Tx data, Length 64
************************************
Start LT_IFFT function
************************************
Complex float pushpacket received, length 64
Processing LT IFFT
IFFT time output (Magnitude), Length 64
LT_IFFT Tx data, Length 128
************************************
Start LT_cyclicPrefix function
************************************
Complex float pushpacket received, length 128
Processing cyclic prefix
LT cyclic prefix modulated data, Length 160
LT_cyclicPrefix Tx data, Length 160
************************************
Start Preamble_append function
************************************
Complex float pushpacket received, length 160
Processing long training sequence storage, size: 160
Processing preamble sequence, size: 320
Preamble_append Tx data, Length 320
************************************
Start PPDU_append function
************************************
Complex float pushpacket received, length 320
Processing Preamble sequence, size: 320
Sent Header control bits
CB Tx data, Length 8
************************************
Start SIGNAL_mapping function
************************************
Real short pushpacket received, length 8
```

```
Processing Header sequence, size: 24
Enter the Rate of the data bits: (Mbit/s) 36
Enter the no. of PSDU Octets to be transmitted: 100
Processing SIGNAL bits, size: 24
SIG Bits, Length 24
**********************************
Start SIGNAL_encoding function
**********************************
Real short pushpacket received, length 24
Processing SIG convolution encoding, size: 24
SIG encoded Tx data, Length 48
**********************************
Start SIGNAL_interleaver function
**********************************
Real short pushpacket received, length 48
Processing SIG interleaver, size: 48
SIG interleaved Tx data, Length 48
**********************************
Start SIG_modulation function
**********************************
Real short pushpacket received, length 48
Processing BPSK modulation
SIG modulated data, Length 48
**********************************
Start SIG_carriers_map function
**********************************
Complex float pushpacket received, length 48
Processing carrier mapping
SIG carriers mapped data, Length 64
**********************************
Start SIG_IFFT function
**********************************
Complex float pushpacket received, length 64
Processing SIG IFFT
IFFT time output (Magnitude), Length 64
SIG_IFFT Tx data, Length 64
**********************************
Start SIG_cyclicPrefix function
**********************************
Complex float pushpacket received, length 64
Processing SIG cyclic prefix
SIG_cyclicPrefix Tx data, Length 80
**********************************
Start SIGNAL_append function
**********************************
Complex float pushpacket received, length 80
Processing SIGNAL sequence, size: 80
Header_append Tx data, Length 96
**********************************
Start PPDU_append function
**********************************
Complex float pushpacket received, length 96
Processing Header sequence, size: 96
Sent DATA control bits
CB Tx data, Length 24
**********************************
Start Data_mapping function
**********************************
Real short pushpacket received, length 24
Processing Data sequence, rate: 36 Mbit/s, length: 100 octets
Activate PSDU processing
Data Tx Bits, Length 24
**********************************
Start Data_PSDU function
**********************************
Real short pushpacket received, length 24
Start PSDU retrieval
PSDU Tx Bits, Length 800
**********************************
Start Data_mapping function
**********************************
```

```
Real short pushpacket received, length 800
Retrieve PSDU data
Append and form raw data packet
Send raw data for scrambling
Data Tx Bits, Length 869
************************************
Start Data_scrambler function
************************************
Real short pushpacket received, length 869
Processing data in the scrambler
Scrambled data Bits, Length 869
************************************
Start Data_tail_replacement function
************************************
Real short pushpacket received, length 869
Processing tail replacement
Tail replaced data Bits, Length 869
************************************
Start Data_encoding function
************************************
Real short pushpacket received, length 869
Processing Data convolution encoding, size: 864
Puncturing the encoded Data, size: 1152
Data encoded Tx data, Length 1157
************************************
Start Data_interleaver function
************************************
Real short pushpacket received, length 1157
Processing Data interleaver, size: 1152
Data interleaved Tx bits, Length 1157
************************************
Start Data_modulation function
************************************
Real short pushpacket received, length 1157
Processing data modulation
Data modulated samples, Length 293
************************************
Start Data_carriers_map function
************************************
Complex float pushpacket received, length 293
Processing carrier mapping
Data carriers mapped data, Length 385
************************************
Start Data_IFFT function
************************************
Complex float pushpacket received, length 385
Processing Data IFFT
IFFT time output (Magnitude), Length 384
Data_IFFT Tx data, Length 385
************************************
Start Data_cyclicPrefix function
************************************
Complex float pushpacket received, length 385
Processing Data cyclic prefix
Data_cyclicPrefix Tx data, Length 480
************************************
Start Data_append function
************************************
Complex float pushpacket received, length 480
Retrieve processed Data samples
Send processed data to form PPDU
Data_append Tx data, Length 480
************************************
Start PPDU_append function
************************************
Complex float pushpacket received, length 480
Processing Data sequence, size: 480
Processing PPDU sequence, size: 880
PPDU octect 0: 0.0459988 + 0.0459988j; -0.132444 + 0.00233959j; -0.0134727 + -0.0785248j;
0.142755 + -0.0126512j; 0.0919975 + 6.18545e-09j; 0.142755 + -0.0126512j; -0.0134727 + -
0.0785248j; -0.132444 + 0.00233958j;
```

PPDU octect 1: 0.0459988 + 0.0459988j; 0.0023396 + -0.132444j; -0.0785248 + -0.0134727j;
-0.0126512 + 0.142755j; 4.31489e-09 + 0.0919975j; -0.0126512 + 0.142755j; -0.0785248 + -
0.0134727j; 0.00233958 + -0.132444j;
PPDU octect 2: 0.0459988 + 0.0459988j; -0.132444 + 0.00233959j; -0.0134727 + -0.0785248j;
0.142755 + -0.0126512j; 0.0919975 + 6.18545e-09j; 0.142755 + -0.0126512j; -0.0134727 + -
0.0785248j; -0.132444 + 0.00233958j;
PPDU octect 3: 0.0459988 + 0.0459988j; 0.0023396 + -0.132444j; -0.0785248 + -0.0134727j;
-0.0126512 + 0.142755j; 4.31489e-09 + 0.0919975j; -0.0126512 + 0.142755j; -0.0785248 + -
0.0134727j; 0.00233958 + -0.132444j;
PPDU octect 4: 0.0459988 + 0.0459988j; -0.132444 + 0.00233959j; -0.0134727 + -0.0785248j;
0.142755 + -0.0126512j; 0.0919975 + 6.18545e-09j; 0.142755 + -0.0126512j; -0.0134727 + -
0.0785248j; -0.132444 + 0.00233958j;
PPDU octect 5: 0.0459988 + 0.0459988j; 0.0023396 + -0.132444j; -0.0785248 + -0.0134727j;
-0.0126512 + 0.142755j; 4.31489e-09 + 0.0919975j; -0.0126512 + 0.142755j; -0.0785248 + -
0.0134727j; 0.00233958 + -0.132444j;
PPDU octect 6: 0.0459988 + 0.0459988j; -0.132444 + 0.00233959j; -0.0134727 + -0.0785248j;
0.142755 + -0.0126512j; 0.0919975 + 6.18545e-09j; 0.142755 + -0.0126512j; -0.0134727 + -
0.0785248j; -0.132444 + 0.00233958j;
PPDU octect 7: 0.0459988 + 0.0459988j; 0.0023396 + -0.132444j; -0.0785248 + -0.0134727j;
-0.0126512 + 0.142755j; 4.31489e-09 + 0.0919975j; -0.0126512 + 0.142755j; -0.0785248 + -
0.0134727j; 0.00233958 + -0.132444j;
PPDU octect 8: 0.0459988 + 0.0459988j; -0.132444 + 0.00233959j; -0.0134727 + -0.0785248j;
0.142755 + -0.0126512j; 0.0919975 + 6.18545e-09j; 0.142755 + -0.0126512j; -0.0134727 + -
0.0785248j; -0.132444 + 0.00233958j;
PPDU octect 9: 0.0459988 + 0.0459988j; 0.0023396 + -0.132444j; -0.0785248 + -0.0134727j;
-0.0126512 + 0.142755j; 4.31489e-09 + 0.0919975j; -0.0126512 + 0.142755j; -0.0785248 + -
0.0134727j; 0.00233958 + -0.132444j;
PPDU octect 10: 0.0459988 + 0.0459988j; -0.132444 + 0.00233959j; -0.0134727 + -
0.0785248j; 0.142755 + -0.0126512j; 0.0919975 + 6.18545e-09j; 0.142755 + -0.0126512j; -
0.0134727 + -0.0785248j; -0.132444 + 0.00233958j;
PPDU octect 11: 0.0459988 + 0.0459988j; 0.0023396 + -0.132444j; -0.0785248 + -0.0134727j;
-0.0126512 + 0.142755j; 4.31489e-09 + 0.0919975j; -0.0126512 + 0.142755j; -0.0785248 + -
0.0134727j; 0.00233958 + -0.132444j;
PPDU octect 12: 0.0459988 + 0.0459988j; -0.132444 + 0.00233959j; -0.0134727 + -
0.0785248j; 0.142755 + -0.0126512j; 0.0919975 + 6.18545e-09j; 0.142755 + -0.0126512j; -
0.0134727 + -0.0785248j; -0.132444 + 0.00233958j;
PPDU octect 13: 0.0459988 + 0.0459988j; 0.0023396 + -0.132444j; -0.0785248 + -0.0134727j;
-0.0126512 + 0.142755j; 4.31489e-09 + 0.0919975j; -0.0126512 + 0.142755j; -0.0785248 + -
0.0134727j; 0.00233958 + -0.132444j;
PPDU octect 14: 0.0459988 + 0.0459988j; -0.132444 + 0.00233959j; -0.0134727 + -
0.0785248j; 0.142755 + -0.0126512j; 0.0919975 + 6.18545e-09j; 0.142755 + -0.0126512j; -
0.0134727 + -0.0785248j; -0.132444 + 0.00233958j;
PPDU octect 15: 0.0459988 + 0.0459988j; 0.0023396 + -0.132444j; -0.0785248 + -0.0134727j;
-0.0126512 + 0.142755j; 4.31489e-09 + 0.0919975j; -0.0126512 + 0.142755j; -0.0785248 + -
0.0134727j; 0.00233958 + -0.132444j;
PPDU octect 16: 0.0459988 + 0.0459988j; -0.132444 + 0.00233959j; -0.0134727 + -
0.0785248j; 0.142755 + -0.0126512j; 0.0919975 + 6.18545e-09j; 0.142755 + -0.0126512j; -
0.0134727 + -0.0785248j; -0.132444 + 0.00233958j;
PPDU octect 17: 0.0459988 + 0.0459988j; 0.0023396 + -0.132444j; -0.0785248 + -0.0134727j;
-0.0126512 + 0.142755j; 4.31489e-09 + 0.0919975j; -0.0126512 + 0.142755j; -0.0785248 + -
0.0134727j; 0.00233958 + -0.132444j;
PPDU octect 18: 0.0459988 + 0.0459988j; -0.132444 + 0.00233959j; -0.0134727 + -
0.0785248j; 0.142755 + -0.0126512j; 0.0919975 + 6.18545e-09j; 0.142755 + -0.0126512j; -
0.0134727 + -0.0785248j; -0.132444 + 0.00233958j;
PPDU octect 19: 0.0459988 + 0.0459988j; 0.0023396 + -0.132444j; -0.0785248 + -0.0134727j;
-0.0126512 + 0.142755j; 4.31489e-09 + 0.0919975j; -0.0126512 + 0.142755j; -0.0785248 + -
0.0134727j; 0.00233958 + -0.132444j;
PPDU octect 20: -0.15625 + 0j; 0.0122846 + -0.0975996j; 0.0917165 + -0.105872j; -
0.0918875 + -0.115129j; -0.00280594 + -0.0537743j; 0.0750737 + 0.0740404j; -0.127324 +
0.0205014j; -0.121887 + 0.0165662j;
PPDU octect 21: -0.0350413 + 0.150888j; -0.0564551 + 0.0218039j; -0.0603101 + -
0.0812861j; 0.0695568 + -0.014122j; 0.0822183 + -0.0923565j; -0.131263 + -0.0652272j; -
0.0572063 + -0.0392986j; 0.0369179 + -0.0983441j;
PPDU octect 22: 0.0625 + 0.0625j; 0.119239 + 0.0040956j; -0.0224832 + -0.160657j;
0.0586688 + 0.014939j; 0.0244759 + 0.0585318j; -0.136805 + 0.0473798j; 0.000988971 +
0.115005j; 0.0533377 + -0.00407633j;
PPDU octect 23: 0.0975412 + 0.0258883j; -0.038316 + 0.106171j; -0.115131 + 0.0551805j;
0.0598238 + 0.0877067j; 0.0211118 + -0.0278859j; 0.0968318 + -0.0827979j; 0.0397497 +
0.111158j; -0.00512124 + 0.120325j;

PPDU octect 24: 0.15625 + 0j; -0.00512125 + -0.120325j; 0.0397497 + -0.111158j; 0.0968319 + 0.0827979j; 0.0211118 + 0.0278859j; 0.0598238 + -0.0877068j; -0.115131 + -0.0551805j; -0.038316 + -0.106171j;
PPDU octect 25: 0.0975412 + -0.0258883j; 0.0533377 + 0.00407635j; 0.000988968 + -0.115005j; -0.136805 + -0.0473798j; 0.0244759 + -0.0585318j; 0.0586688 + -0.014939j; -0.0224832 + 0.160657j; 0.119239 + -0.00409556j;
PPDU octect 26: 0.0625 + -0.0625j; 0.0369179 + 0.0983441j; -0.0572063 + 0.0392986j; -0.131263 + 0.0652272j; 0.0822183 + 0.0923565j; 0.0695568 + 0.014122j; -0.0603101 + 0.0812861j; -0.0564551 + -0.0218039j;
PPDU octect 27: -0.0350412 + -0.150888j; -0.121887 + -0.0165662j; -0.127324 + -0.0205014j; 0.0750737 + -0.0740404j; -0.00280595 + 0.0537742j; -0.0918875 + 0.115129j; 0.0917165 + 0.105872j; 0.0122846 + 0.0975995j;
PPDU octect 28: -0.15625 + 0j; 0.0122846 + -0.0975996j; 0.0917165 + -0.105872j; -0.0918875 + -0.115129j; -0.00280594 + -0.0537743j; 0.0750737 + 0.0740404j; -0.127324 + 0.0205014j; -0.121887 + 0.0165662j;
PPDU octect 29: -0.0350413 + 0.150888j; -0.0564551 + 0.0218039j; -0.0603101 + -0.0812861j; 0.0695568 + -0.014122j; 0.0822183 + -0.0923565j; -0.131263 + -0.0652272j; -0.0572063 + -0.0392986j; 0.0369179 + -0.0983441j;
PPDU octect 30: 0.0625 + 0.0625j; 0.119239 + 0.0040956j; -0.0224832 + -0.160657j; 0.0586688 + 0.014939j; 0.0244759 + 0.0585318j; -0.136805 + 0.0473798j; 0.000988971 + 0.115005j; 0.0533377 + -0.00407633j;
PPDU octect 31: 0.0975412 + 0.0258883j; -0.038316 + 0.106171j; -0.115131 + 0.0551805j; 0.0598238 + 0.0877067j; 0.0211118 + -0.0278859j; 0.0968318 + -0.0827979j; 0.0397497 + 0.111158j; -0.00512124 + 0.120325j;
PPDU octect 32: 0.15625 + 0j; -0.00512125 + -0.120325j; 0.0397497 + -0.111158j; 0.0968319 + 0.0827979j; 0.0211118 + 0.0278859j; 0.0598238 + -0.0877068j; -0.115131 + -0.0551805j; -0.038316 + -0.106171j;
PPDU octect 33: 0.0975412 + -0.0258883j; 0.0533377 + 0.00407635j; 0.000988968 + -0.115005j; -0.136805 + -0.0473798j; 0.0244759 + -0.0585318j; 0.0586688 + -0.014939j; -0.0224832 + 0.160657j; 0.119239 + -0.00409556j;
PPDU octect 34: 0.0625 + -0.0625j; 0.0369179 + 0.0983441j; -0.0572063 + 0.0392986j; -0.131263 + 0.0652272j; 0.0822183 + 0.0923565j; 0.0695568 + 0.014122j; -0.0603101 + 0.0812861j; -0.0564551 + -0.0218039j;
PPDU octect 35: -0.0350412 + -0.150888j; -0.121887 + -0.0165662j; -0.127324 + -0.0205014j; 0.0750737 + -0.0740404j; -0.00280595 + 0.0537742j; -0.0918875 + 0.115129j; 0.0917165 + 0.105872j; 0.0122846 + 0.0975995j;
PPDU octect 36: -0.15625 + 0j; 0.0122846 + -0.0975996j; 0.0917165 + -0.105872j; -0.0918875 + -0.115129j; -0.00280594 + -0.0537743j; 0.0750737 + 0.0740404j; -0.127324 + 0.0205014j; -0.121887 + 0.0165662j;
PPDU octect 37: -0.0350413 + 0.150888j; -0.0564551 + 0.0218039j; -0.0603101 + -0.0812861j; 0.0695568 + -0.014122j; 0.0822183 + -0.0923565j; -0.131263 + -0.0652272j; -0.0572063 + -0.0392986j; 0.0369179 + -0.0983441j;
PPDU octect 38: 0.0625 + 0.0625j; 0.119239 + 0.0040956j; -0.0224832 + -0.160657j; 0.0586688 + 0.014939j; 0.0244759 + 0.0585318j; -0.136805 + 0.0473798j; 0.000988971 + 0.115005j; 0.0533377 + -0.00407633j;
PPDU octect 39: 0.0975412 + 0.0258883j; -0.038316 + 0.106171j; -0.115131 + 0.0551805j; 0.0598238 + 0.0877067j; 0.0211118 + -0.0278859j; 0.0968318 + -0.0827979j; 0.0397497 + 0.111158j; -0.00512124 + 0.120325j;
PPDU octect 40: 0.0625 + 0j; 0.0330338 + -0.0438527j; -0.00196556 + -0.037627j; -0.0809046 + 0.0844305j; 0.00677414 + -0.100151j; -0.00124631 + -0.113302j; -0.0211465 + -0.00464019j; 0.135693 + -0.10469j;
PPDU octect 41: 0.0975413 + -0.0441942j; 0.0112075 + -0.00183259j; -0.0327044 + 0.0440799j; -0.0604833 + 0.124227j; 0.0101382 + 0.0966019j; 0.000441049 + -0.00776956j; 0.0183601 + -0.0825037j; -0.0692837 + 0.0267335j;
PPDU octect 42: -0.21875 + 0j; -0.0692838 + -0.0267335j; 0.0183601 + 0.0825037j; 0.000441058 + 0.00776954j; 0.0101382 + -0.0966019j; -0.0604833 + -0.124227j; -0.0327044 + -0.04408j; 0.0112075 + 0.00183258j;
PPDU octect 43: 0.0975413 + 0.0441942j; 0.135693 + 0.10469j; -0.0211466 + 0.00464019j; -0.00124632 + 0.113302j; 0.00677412 + 0.100151j; -0.0809046 + -0.0844305j; -0.00196555 + 0.037627j; 0.0330337 + 0.0438527j;
PPDU octect 44: 0.0625 + 0j; 0.0572128 + 0.0524973j; 0.0155139 + 0.173851j; 0.0354616 + 0.115564j; -0.0509683 + -0.201625j; 0.0107812 + 0.0359063j; 0.0892584 + 0.208513j; -0.0485137 + -0.00788796j;
PPDU octect 45: -0.0350413 + 0.0441942j; 0.0170981 + -0.058969j; 0.0529809 + -0.0169834j; 0.0987838 + 0.100154j; 0.034056 + -0.148379j; -0.00283341 + -0.0940129j; -0.120297 + 0.0419507j; -0.136448 + -0.0698656j;
PPDU octect 46: -0.03125 + 0j; -0.136448 + 0.0698656j; -0.120297 + -0.0419508j; -0.0028334 + 0.0940129j; 0.0340559 + 0.148379j; 0.0987838 + -0.100154j; 0.0529809 + 0.0169834j; 0.0170981 + 0.058969j;

PPDU octect 47: -0.0350413 + -0.0441942j; -0.0485138 + 0.00788795j; 0.0892585 + -0.208513j; 0.0107813 + -0.0359063j; -0.0509683 + 0.201625j; 0.0354616 + -0.115564j; 0.0155138 + -0.173851j; 0.0572128 + -0.0524973j;
PPDU octect 48: 0.0625 + 0j; 0.0330338 + -0.0438527j; -0.00196556 + -0.037627j; -0.0809046 + 0.0844305j; 0.00677414 + -0.100151j; -0.00124631 + -0.113302j; -0.0211465 + -0.00464019j; 0.135693 + -0.10469j;
PPDU octect 49: 0.0975413 + -0.0441942j; 0.0112075 + -0.00183259j; -0.0327044 + 0.0440799j; -0.0604833 + 0.124227j; 0.0101382 + 0.0966019j; 0.000441049 + -0.00776956j; 0.0183601 + -0.0825037j; -0.0692837 + 0.0267335j;
PPDU octect 50: -0.0592927 + 0.100425j; 0.00409242 + 0.0139711j; 0.0109037 + -0.100262j; -0.0969928 + -0.0203457j; 0.0621235 + 0.08138j; 0.123595 + 0.138917j; 0.104256 + -0.0150552j; 0.172935 + -0.139791j;
PPDU octect 51: -0.0395934 + 0.00585377j; -0.133488 + 0.00892884j; -0.00158766 + -0.0432846j; -0.0472771 + 0.0921656j; -0.109013 + 0.0817054j; -0.0239633 + 0.0104072j; 0.0964961 + 0.0185489j; 0.0191099 + -0.0225715j;
PPDU octect 52: -0.0873354 + -0.0494106j; 0.00234121 + 0.0581253j; -0.0210441 + 0.228457j; -0.102889 + 0.0228245j; -0.0192593 + -0.175154j; 0.0178246 + 0.131774j; -0.0710194 + 0.16038j; -0.15319 + -0.0619258j;
PPDU octect 53: -0.107073 + 0.0278859j; 0.055435 + 0.140018j; 0.069911 + 0.102684j; -0.0555579 + 0.0249016j; -0.0427567 + 0.00162033j; 0.0156848 + -0.118058j; 0.0255435 + -0.0712484j; 0.0332799 + 0.177229j;
PPDU octect 54: 0.0197642 + -0.0213679j; 0.0353311 + -0.0884221j; -0.00812379 + 0.100698j; -0.0349064 + -0.00964246j; 0.0646485 + 0.0304234j; 0.0924734 + -0.0338842j; 0.0319434 + -0.122535j; -0.0175289 + 0.0916916j;
PPDU octect 55: 6.49395e-05 + -0.00585376j; -0.00622475 + -0.0561029j; -0.019328 + 0.0397577j; 0.0532 + -0.131378j; 0.021769 + -0.132811j; 0.104157 + -0.0317j; 0.162671 + -0.0445116j; -0.10486 + -0.029587j;
PPDU octect 56: -0.110307 + -0.0691748j; -0.00773789 + -0.0918839j; -0.0492151 + -0.0428284j; 0.0847025 + -0.0172379j; 0.0901296 + 0.0633508j; 0.01488 + 0.153376j; 0.0486094 + 0.0938087j; 0.011148 + 0.0339775j;
PPDU octect 57: -0.0115127 + 0.0116426j; -0.0152419 + -0.0173805j; -0.0605729 + 0.0310065j; -0.0701968 + -0.0403441j; 0.0114148 + -0.108628j; 0.03707 + -0.0599709j; -0.00321576 + -0.177502j; -0.00720509 + -0.128079j;
PPDU octect 58: -0.0592927 + 0.100425j; 0.00409242 + 0.0139711j; 0.0109037 + -0.100262j; -0.0969928 + -0.0203457j; 0.0621235 + 0.08138j; 0.123595 + 0.138917j; 0.104256 + -0.0150552j; 0.172935 + -0.139791j;
PPDU octect 59: -0.0395934 + 0.00585377j; -0.133488 + 0.00892884j; -0.00158766 + -0.0432846j; -0.0472771 + 0.0921656j; -0.109013 + 0.0817054j; -0.0239633 + 0.0104072j; 0.0964961 + 0.0185489j; 0.0191099 + -0.0225715j;
PPDU octect 60: -0.0296464 + 0.0806606j; -0.0963844 + -0.0450347j; -0.1102 + 0.00288139j; -0.0700095 + 0.215747j; -0.0396332 + 0.0593958j; 0.00997518 + -0.0555177j; 0.0337749 + 0.0652693j; 0.11684 + 0.0332285j;
PPDU octect 61: 0.0779988 + -0.133198j; -0.0427866 + -0.146173j; 0.158107 + -0.0705024j; 0.253711 + -0.0210639j; 0.067816 + 0.116981j; -0.0441807 + 0.114255j; -0.0354926 + 0.0410436j; 0.0845137 + 0.0701878j;
PPDU octect 62: 0.120189 + 0.00988212j; 0.0573284 + 0.0546368j; 0.0632006 + 0.187834j; 0.0906149 + 0.149391j; -0.0165717 + -0.0393876j; -0.0775449 + -0.0749532j; 0.0494644 + 0.0792623j; -0.0139247 + -0.00728012j;
PPDU octect 63: 0.0302837 + -0.0273135j; 0.080169 + 0.0537303j; -0.185944 + -0.0667172j; -0.0386776 + -0.0274393j; 0.0430363 + -0.0718034j; -0.0919199 + -0.089421j; 0.0290079 + 0.105391j; -0.144236 + 0.0033899j;
PPDU octect 64: -0.0691748 + -0.0411321j; 0.131827 + 0.0566497j; -0.126439 + 0.0697465j; -0.0308446 + 0.108688j; 0.160616 + -0.00928296j; 0.0555444 + -0.0462641j; -0.00366729 + 0.0278459j; -0.0492698 + 0.000149147j;
PPDU octect 65: -0.0779988 + -0.00515146j; 0.0145183 + -0.0870846j; 0.148909 + -0.103997j; -0.0212099 + -0.0514882j; -0.154066 + -0.106397j; 0.02395 + 0.0303362j; 0.0463352 + 0.122983j; -0.00377208 + -0.0983907j;
PPDU octect 66: -0.0608964 + -0.128468j; -0.0236949 + -0.0380531j; 0.0664307 + -0.0484323j; -0.0671134 + 0.0266201j; 0.0537023 + -0.0502537j; 0.170927 + -0.0486926j; -0.107523 + 0.132273j; -0.161486 + -0.0194366j;
PPDU octect 67: -0.0698122 + -0.0715077j; -0.176879 + 0.0491159j; -0.172178 + -0.0498698j; 0.0512337 + -0.0746361j; 0.122271 + -0.0573664j; 0.00915009 + -0.0437585j; -0.0118996 + -0.0206696j; 0.00363008 + 0.00856091j;
PPDU octect 68: -0.0296464 + 0.0806606j; -0.0963844 + -0.0450347j; -0.1102 + 0.00288139j; -0.0700095 + 0.215747j; -0.0396332 + 0.0593958j; 0.00997518 + -0.0555177j; 0.0337749 + 0.0652693j; 0.11684 + 0.0332285j;
PPDU octect 69: 0.0779988 + -0.133198j; -0.0427866 + -0.146173j; 0.158107 + -0.0705024j; 0.253711 + -0.0210639j; 0.067816 + 0.116981j; -0.0441807 + 0.114255j; -0.0354926 + 0.0410436j; 0.0845137 + 0.0701878j;

PPDU octect 70: -0.118585 + 0.0114858j; -0.0994333 + -0.047962j; 0.0536084 + -0.196079j;
0.123994 + 0.0345825j; 0.0919315 + 0.0449895j; -0.036786 + -0.0658418j; -0.0211003 + -
0.00387933j; 0.0424973 + -0.0649011j;
PPDU octect 71: 0.0611289 + 0.0482766j; 0.0463064 + 0.00417344j; -0.0628931 + -
0.0452457j; -0.101784 + 0.152274j; -0.0392474 + -0.0187027j; -0.00526792 + -0.106066j;
0.0827078 + 0.0305796j; 0.225664 + 0.0276615j;
PPDU octect 72: 0.139953 + -0.00988211j; -0.132354 + -0.0329092j; -0.116179 + 0.0883309j;
0.0227295 + 0.0518396j; -0.171248 + -0.0803886j; -0.245748 + -0.0245908j; -0.0624398 + -
0.0379207j; -0.0549765 + -0.06221j;
PPDU octect 73: -0.00395256 + -0.0598542j; 0.0338285 + -1.12667e-05j; -0.0302122 +
0.0214911j; 0.0747573 + -0.121648j; 0.04317 + -0.0796105j; -0.0224231 + 0.0414547j;
0.0263911 + 0.0131129j; -0.03097 + -0.0184544j;
PPDU octect 74: 0.0592927 + 0.00827848j; 0.108935 + 0.0779861j; 0.00206425 + 0.101085j; -
0.0158528 + 0.0540659j; -0.059185 + 0.0702049j; 0.0168651 + 0.114138j; 0.103643 + -
0.0339088j; -0.0242415 + -0.0589005j;
PPDU octect 75: -0.0808932 + 0.0505445j; -0.0402822 + -0.0687858j; -0.0685797 +
0.0580919j; -0.067265 + 0.117231j; 0.00650102 + -0.131225j; 0.00858317 + 0.0280095j;
0.075177 + 0.116767j; 0.117563 + 0.0296694j;
PPDU octect 76: -0.0411321 + 0.148232j; 0.00497321 + 0.0976094j; 0.0257738 + 0.00186785j;
-0.115934 + 0.0446468j; -0.0196118 + 0.0837796j; 0.100749 + 0.00625553j; 0.20549 + -
0.0640454j; 0.0729341 + -0.0633013j;
PPDU octect 77: -0.173926 + -0.118024j; -0.0241854 + 0.0258011j; -0.040753 + 0.128572j; -
0.0420396 + -0.0534839j; 0.14769 + -0.126218j; -0.0299877 + -0.0492609j; -0.0145844 + -
0.0207055j; 0.0891527 + -0.0690722j;
PPDU octect 78: -0.118585 + 0.0114858j; -0.0994333 + -0.047962j; 0.0536084 + -0.196079j;
0.123994 + 0.0345825j; 0.0919315 + 0.0449895j; -0.036786 + -0.0658418j; -0.0211003 + -
0.00387933j; 0.0424973 + -0.0649011j;
PPDU octect 79: 0.0611289 + 0.0482766j; 0.0463064 + 0.00417344j; -0.0628931 + -
0.0452457j; -0.101784 + 0.152274j; -0.0392474 + -0.0187027j; -0.00526792 + -0.106066j;
0.0827078 + 0.0305796j; 0.225664 + 0.0276615j;
PPDU octect 80: 0.0296464 + -0.120189j; 0.0340277 + -0.142461j; 0.00366307 + -0.0123123j;
0.12597 + -0.0429729j; 0.0545228 + 0.0680213j; -0.0196314 + 0.0772427j; 0.00787072 + -
0.0556209j; -0.0343327 + 0.0461655j;
PPDU octect 81: -0.0396692 + -0.133825j; -0.056498 + -0.131124j; 0.0143079 + 0.0966673j;
0.0448767 + -0.00858707j; -0.112614 + -0.17049j; -0.065255 + -0.229676j; 0.0651514 + -
0.0114664j; 0.011351 + 0.047573j;
PPDU octect 82: -0.0905427 + -0.0592927j; -0.109871 + 0.024421j; 0.0738487 + -0.0343458j;
0.124333 + 0.0215448j; -0.0371767 + 0.0707535j; 0.0153654 + 0.00152866j; 0.0280382 +
0.099403j; -0.0620882 + 0.0682245j;
PPDU octect 83: 0.0639476 + 0.0162325j; 0.0781643 + 0.156022j; 0.00886139 + 0.219155j;
0.146588 + 0.0238822j; 0.105724 + 0.0303761j; -0.0804061 + 0.142788j; -0.048684 + -
0.0996885j; -0.0360851 + -0.0822708j;
PPDU octect 84: -0.0889391 + 0.0213679j; -0.0700184 + -0.0293556j; -0.0862983 +
0.0483206j; -0.0657101 + -0.0154747j; -0.0241742 + 0.00185581j; -0.0304487 + -0.0230262j;
-0.0317125 + 0.0199536j; -0.00206234 + 0.211811j;
PPDU octect 85: 0.158255 + -0.0242892j; 0.141453 + -0.118609j; -0.146111 + 0.0575272j; -
0.155152 + 0.0833329j; -0.0015876 + -0.0295499j; 0.018425 + -0.129305j; 0.012172 + -
0.0180522j; -0.0083002 + -0.0371899j;
PPDU octect 86: 0.03125 + 0.0395285j; 0.0234214 + 0.0965821j; 0.0135822 + -0.0392045j;
0.0498857 + 0.0189327j; -0.0722289 + -0.140631j; -0.0228469 + -0.0508914j; 0.0237545 +
0.0991738j; -0.127134 + -0.116193j;
PPDU octect 87: 0.0941663 + 0.102353j; 0.182928 + 0.0982142j; -0.0399674 + -0.0195794j;
0.0646191 + 0.077456j; 0.0875343 + -0.146564j; -0.038809 + -0.0585788j; -0.0565902 +
0.124412j; -0.0767586 + 0.0199933j;
PPDU octect 88: 0.0296464 + -0.120189j; 0.0340277 + -0.142461j; 0.00366307 + -0.0123123j;
0.12597 + -0.0429729j; 0.0545228 + 0.0680213j; -0.0196314 + 0.0772427j; 0.00787072 + -
0.0556209j; -0.0343327 + 0.0461655j;
PPDU octect 89: -0.0396692 + -0.133825j; -0.056498 + -0.131124j; 0.0143079 + 0.0966673j;
0.0448767 + -0.00858707j; -0.112614 + -0.17049j; -0.065255 + -0.229676j; 0.0651514 + -
0.0114664j; 0.011351 + 0.047573j;
PPDU octect 90: 0.0395285 + 0.0181606j; -0.00163598 + 0.0412795j; 0.00138841 +
0.0708434j; -0.0373389 + -0.116939j; -0.105664 + -0.0623j; 0.00181488 + 0.0568087j; -
0.0084312 + -0.0109701j; 0.0187883 + 0.0721539j;
PPDU octect 91: 0.0162325 + 0.0587311j; -0.0652257 + -0.0766981j; 0.141542 + -0.0617811j;
0.0869275 + 0.025459j; -0.00262285 + -0.102865j; 0.106662 + -0.151683j; -0.0544175 +
0.036349j; -0.0296711 + -0.00309548j;
PPDU octect 92: 0.0576891 + -0.0197642j; -0.0278851 + 0.0066721j; -0.027348 + -
0.0986786j; 0.0487899 + -0.0752348j; 0.174449 + 0.0308429j; 0.133943 + 0.155555j;
0.0604381 + 0.076652j; -0.0104721 + -0.0218301j;

```
PPDU octect 93: -0.0835819 + 0.04009j; -0.073946 + 0.0110531j; -0.163122 + 0.0540938j; -
0.0520085 + -0.00831897j; 0.0762527 + -0.0419279j; 0.0425213 + 0.100944j; 0.0576176 + -
0.0183252j; 0.00311136 + -0.0899109j;
PPDU octect 94: 0.0592927 + -0.0181606j; 0.0230726 + -0.0309362j; 0.00711971 + -
0.0172156j; 0.066082 + -0.016897j; -0.13531 + -0.0982117j; -0.0556543 + -0.0807799j;
0.0885568 + 0.154351j; 0.119511 + 0.122336j;
PPDU octect 95: 0.102353 + 0.000561578j; -0.141042 + 0.102104j; 0.0063104 + -0.0114042j;
0.0566463 + -0.0393582j; -0.0590676 + 0.065734j; 0.131893 + 0.111102j; 0.0119852 +
0.113766j; 0.0468643 + -0.105996j;
PPDU octect 96: 0.159718 + -0.0988212j; -0.076392 + 0.0844907j; -0.0486395 + 0.0730016j;
0.00507324 + -0.0861391j; -0.0520616 + -0.107502j; -0.0726498 + 0.128766j; -0.128986 + -
0.0339684j; -0.15277 + -0.11095j;
PPDU octect 97: -0.193117 + 0.0982596j; -0.107315 + -0.0684993j; 0.00369191 + -
0.00885941j; -0.0392129 + 0.0243554j; -0.0540907 + -0.0790553j; 0.023725 + 0.0841624j;
0.052294 + -0.00162637j; 0.0277925 + -0.0439759j;
PPDU octect 98: 0.0395285 + 0.0181606j; -0.00163598 + 0.0412795j; 0.00138841 +
0.0708434j; -0.0373389 + -0.116939j; -0.105664 + -0.0623j; 0.00181488 + 0.0568087j; -
0.0084312 + -0.0109701j; 0.0187883 + 0.0721539j;
PPDU octect 99: 0.0162325 + 0.0587311j; -0.0652257 + -0.0766981j; 0.141542 + -0.0617811j;
0.0869275 + 0.025459j; -0.00262285 + -0.102865j; 0.106662 + -0.151683j; -0.0544175 +
0.036349j; -0.0296711 + -0.00309548j;
PPDU octect 100: 0.0197642 + -0.159718j; 0.0290886 + 0.0252547j; 0.0861483 + -0.0288349j;
0.0867942 + -0.0815142j; 0.00345816 + -0.0359947j; -0.0960365 + -0.0885953j; -0.0729433 +
-0.0459031j; 0.105487 + -0.0196636j;
PPDU octect 101: 0.192621 + 0.0180038j; -0.0531459 + -0.073101j; -0.118339 + -0.148885j;
0.0191895 + -0.0190929j; -0.0417141 + 0.0263756j; 0.0405524 + 0.00882431j; 0.0284543 + -
0.0764756j; -0.0376039 + -0.0684193j;
PPDU octect 102: -0.0114858 + 0.00988212j; -0.133529 + -0.0644879j; 0.0689293 + -
0.0671722j; 0.0570357 + 0.00630021j; -0.134364 + 0.0980107j; 0.152477 + 0.0360762j;
0.0412595 + -0.0845433j; -0.099111 + -0.0485833j;
PPDU octect 103: 0.089004 + -0.0994585j; -0.0459865 + 0.0181825j; -0.112344 + 0.135481j;
-0.0635739 + 0.0178506j; -0.0222531 + 0.0530081j; 0.0410498 + 0.0765494j; -0.0211672 +
0.144644j; 0.00665332 + 0.178782j;
PPDU octect 104: 0.0592927 + 0.0411321j; 0.0225525 + 0.0644587j; 0.0616463 + 0.0217485j;
0.110484 + -0.0806779j; -0.016029 + -0.0536466j; -0.0138699 + -0.0174774j; 0.171277 +
0.00807664j; 0.0701664 + -0.026787j;
PPDU octect 105: -0.0147426 + 0.00176044j; -0.0124264 + 0.0529065j; -0.125061 +
0.00871965j; -0.039783 + 0.0123833j; 0.0359253 + 0.114372j; 0.00701687 + 0.0898629j; -
0.0158081 + -0.0822487j; -0.00846144 + -0.0129339j;
PPDU octect 106: 0.0905427 + 0.0296463j; 0.0723212 + -0.0683032j; 0.0512069 + 0.0626811j;
-0.0042373 + 0.0486363j; -0.129764 + -0.0478978j; -0.120922 + 0.0614277j; -0.0952694 +
0.0780457j; 0.0106693 + 0.00456515j;
PPDU octect 107: 0.0493456 + 0.000637334j; -0.0138327 + -0.0108121j; 0.00875645 + -
0.0627949j; -0.030951 + 0.0402129j; -0.0114865 + 0.00388678j; -0.0334234 + -0.110766j; -
0.11486 + 0.137461j; -0.0246448 + 0.0489417j;
PPDU octect 108: 0.0197642 + -0.159718j; 0.0290886 + 0.0252547j; 0.0861483 + -0.0288349j;
0.0867942 + -0.0815142j; 0.00345816 + -0.0359947j; -0.0960365 + -0.0885953j; -0.0729433 +
-0.0459031j; 0.105487 + -0.0196636j;
PPDU octect 109: 0.192621 + 0.0180038j; -0.0531459 + -0.073101j; -0.118339 + -0.148885j;
0.0191895 + -0.0190929j; -0.0417141 + 0.0263756j; 0.0405524 + 0.00882431j; 0.0284543 + -
0.0764756j; -0.0376039 + -0.0684193j;
**********************************
Phew! It finally works ... ...
**********************************
End of OSSIE simulation: 802.11a PPDU transmission
**********************************
```

B. RECEIVER

```
************************************
Welcome to OSSIE simulation: 802.11a PPDU receiver
************************************
Start PPDU digitised receiver
stream_sizeI = 884, stream_sizeQ = 884
Removed preamble from PPDU samples
Preamble match at pointer: 4
Sent PPDU (preamble removed) samples
PPDU_preamble_removed data, Length 560
************************************
Start PPDU Header remove function
************************************
Complex float pushpacket received, length 560
Sent SIG time samples for processing ...
Header removed data output, Length 80
************************************
Start SIG_cyclicPrefix_rem function
************************************
Complex float pushpacket received, length 80
Processing SIG cyclic prefix remove
SIG_cyclicPrefix_rem Tx data, Length 64
************************************
Start SIG_FFT function
************************************
Complex float pushpacket received, length 64
Processing SIG FFT
FFT frequency output (Magnitude), Length 64
SIG_FFT Tx data, Length 64
************************************
Start SIG_carriers_demap function
************************************
Complex float pushpacket received, length 64
Processing carrier demapping
SIG carriers demapped data, Length 48
************************************
Start SIG_demodulation function
************************************
Complex float pushpacket received, length 48
Processing SIG BPSK demodulation
SIG demodulated data, Length 48
************************************
Start SIG_deinterleaver function
************************************
Real short pushpacket received, length 48
Processing SIG deinterleaver, size: 48
SIG deinterleaved Tx data, Length 48
************************************
Start SIG convolution decoding function
************************************
Real short pushpacket received, length 48
Processing SIG convolution decoding, size: 48
We are inside initialise_viterbi
Nstate = 64; Nsr = 6
return from initialise viterbi
We are inside process_viterbi
No. of iterations: 24
return from process viterbi
End of decoding
SIG decoded Tx data, Length 24
************************************
SIG header information received
************************************
Real short pushpacket received, length 24
Processing Data sequence, rate: 36 Mbit/s, length: 100 octets
Send raw data for decoding
Header removed data output, Length 485
************************************
Start Data recovery function
************************************
```

```
Complex float pushpacket received, length 485
Sent raw data for processing ...
Data output, Length 485
************************************
Start Data_cyclicPrefix_rem function
************************************
Complex float pushpacket received, length 485
Processing Data cyclic prefix remove
Data_cyclicPrefix_rem Tx data, Length 389
************************************
Start Data_FFT function
************************************
Complex float pushpacket received, length 389
Processing Data FFT
FFT frequency output (Magnitude), Length 384
Data_FFT Tx data, Length 389
************************************
Start Data_carriers_demap function
************************************
Complex float pushpacket received, length 389
Processing data carrier demapping
Data carriers demapped data, Length 293
************************************
Start Data_demodulation function
************************************
Complex float pushpacket received, length 293
Processing Data demodulation
Conduct 16QAM demodulation
Data demodulated data, Length 1157
************************************
Start Data_deinterleaver function
************************************
Real short pushpacket received, length 1157
Processing Data deinterleaver
Data deinterleaved Tx data, Length 1157
************************************
Start Data convolution decoding function
************************************
Real short pushpacket received, length 1157
Processing Data convolution decoding
Recover the punctured encoded Data, size: 1152
We are inside initialise_viterbi
Nstate = 64; Nsr = 6
return from initialise viterbi
We are inside process_viterbi
No. of iterations: 144
We are inside process_viterbi
No. of iterations: 144
We are inside process_viterbi
No. of iterations: 144
We are inside process_viterbi
No. of iterations: 144
We are inside process_viterbi
No. of iterations: 144
We are inside process_viterbi
No. of iterations: 144
We are inside process_viterbi
No. of iterations: 144
return from process viterbi
End of decoding
Data decoded Tx data, Length 869
************************************
Start Data_descrambler function
************************************
Real short pushpacket received, length 869
Processing data in the descrambler
Descrambled data Bits, Length 864
************************************
Decoded Data received
************************************
Real short pushpacket received, length 864
Send PSDU data to MAC layer ...
PSDU Data output, Length 801
```

```
**********************************
Start Data_PSDU display function
**********************************
Real short pushpacket received, length 801
Start PSDU display ...
Received data successfully decoded:
Received octet data [0]: 04
Received octet data [1]: 02
Received octet data [2]: 00
Received octet data [3]: 2e
Received octet data [4]: 00
Received octet data [5]: 60
Received octet data [6]: 08
Received octet data [7]: cd
Received octet data [8]: 37
Received octet data [9]: a6
Received octet data [10]: 00
Received octet data [11]: 20
Received octet data [12]: d6
Received octet data [13]: 01
Received octet data [14]: 3c
Received octet data [15]: f1
Received octet data [16]: ef
Received octet data [17]: 7a
Received octet data [18]: 51
Received octet data [19]: 92
Received octet data [20]: 3b
Received octet data [21]: af
Received octet data [22]: 00
Received octet data [23]: 00
Received octet data [24]: 4a
Received octet data [25]: 6f
Received octet data [26]: 79
Received octet data [27]: 2c
Received octet data [28]: 20
Received octet data [29]: 62
Received octet data [30]: 72
Received octet data [31]: 69
Received octet data [32]: 67
Received octet data [33]: 68
Received octet data [34]: a3
Received octet data [35]: 2f
Received octet data [36]: 73
Received octet data [37]: 70
Received octet data [38]: 61
Received octet data [39]: 72
Received octet data [40]: 6b
Received octet data [41]: 20
Received octet data [42]: 6f
Received octet data [43]: 66
Received octet data [44]: 20
Received octet data [45]: 64
Received octet data [46]: 69
Received octet data [47]: 76
Received octet data [48]: 69
Received octet data [49]: 6e
Received octet data [50]: 69
Received octet data [51]: 74
Received octet data [52]: 83
Received octet data [53]: 2d
Received octet data [54]: 0a
Received octet data [55]: 44
Received octet data [56]: 61
Received octet data [57]: 75
Received octet data [58]: 67
Received octet data [59]: 68
Received octet data [60]: 74
Received octet data [61]: 65
Received octet data [62]: 72
Received octet data [63]: 20
Received octet data [64]: 6f
```

```
Received octet data [65]: 66
Received octet data [66]: 20
Received octet data [67]: 45
Received octet data [68]: 6c
Received octet data [69]: 79
Received octet data [70]: 72
Received octet data [71]: 69
Received octet data [72]: 75
Received octet data [73]: 6d
Received octet data [74]: 2c
Received octet data [75]: 0a
Received octet data [76]: 46
Received octet data [77]: 69
Received octet data [78]: 72
Received octet data [79]: 65
Received octet data [80]: 2d
Received octet data [81]: 69
Received octet data [82]: 6e
Received octet data [83]: 73
Received octet data [84]: 69
Received octet data [85]: 72
Received octet data [86]: 65
Received octet data [87]: 64
Received octet data [88]: 20
Received octet data [89]: 77
Received octet data [90]: 65
Received octet data [91]: 20
Received octet data [92]: 74
Received octet data [93]: 72
Received octet data [94]: 65
Received octet data [95]: 61
Received octet data [96]: da
Received octet data [97]: 57
Received octet data [98]: 99
Received octet data [99]: ed
**********************************
Phew! It finally works ... ...
**********************************
End of OSSIE simulation: 802.11a PPDU receiver
**********************************
```

APPENDIX D: IEEE 802.11A TEST SEQUENTIAL FLOW CHART

A. TRANSMITTER

	IEEE 802.11a Transmitter components sequential flow						Example (802.11a Annex G)	
	Components	**Description**	**Data In**	**Type In**	**Data Out**	**Type Out**	**Data size in**	**Data size out**
1	**preamble_map**	- initiate the tx routine - start ST sequence processing	-	-	complex	integer	-	52
2	**ST_carrier_map**	- ST carrier mapping	complex	integer	complex	float	52	64
3	**ST_IFFT**	- ST IFFT	complex	float	complex	float	64	160
4	**preamble_map**	- receive ST samples - start LT sequence processing	complex	float	complex	integer	160	52
5	**LT_carrier_map**	- LT carrier mapping	complex	integer	complex	float	52	64
6	**LT_IFFT**	- LT IFFT	complex	float	complex	float	64	128
7	**LT_cyclicPrefix**	- LT cyclic prefix append	complex	float	complex	float	128	160
8	**preamble_map**	- receive LT samples - send preamble (ST + LT) to PPDU	complex	float	complex	float	160	320
9	**PPDU_map**	- receive preamble - send CB to start SIG processing	complex	float	real	integer	320	8
10	**header_map**	- receive RATE & LENGTH - send raw SIG for processing	real	integer	real	integer	8	24
11	**SIG_conv_enc**	- SIG convolution encoding	real	integer	real	integer	24	48
12	**SIG_interleaver**	- SIG interleaving	real	integer	real	integer	48	48
13	**SIG_BPSK_mod**	- SIG BPSK modulation	real	integer	complex	float	48	48
14	**SIG_carriers_map**	- SIG carriers mapping	complex	float	complex	float	48	64
15	**SIG_IFFT**	- SIG IFFT	complex	float	complex	float	64	64
16	**SIG_cyclicprefix**	- SIG cyclic prefix	complex	float	complex	float	64	80

	IEEE 802.11a Transmitter components sequential flow						Example (802.11a Annex G)	
	Components	**Description**	**Data In**	**Type In**	**Data Out**	**Type Out**	**Data size in**	**Data size out**
17	**header_map**	- receive SIG samples - send SIG to PPDU	complex	float	complex	float	80	96
18	**PPDU_map**	- receive SIG samples - send CB to start data processing	complex	float	real	integer	96	24
19	**data_map**	- receive RATE & LENGTH - send CB to start PSDU processing	real	integer	real	integer	24	24
20	**data_PSDU**	- input PSDU data	real	integer	real	integer	24	800
21	**data_map**	- receive PSDU data - send CB to start PSDU processing	real	integer	real	integer	800	869
22	**data_scrambler**	- scrambler the raw data	real	integer	real	integer	869	869
23	**data_tail_replacement**	- replace tail with zeroes	real	integer	real	integer	869	869
24	**data_conv_enc**	- data convolution encoding - data puncturing	real	integer	real	integer	869	1157
25	**data_interleaver**	- data interleaving	real	integer	real	integer	1157	1157
26	**data_mod_map**	- data modulation mapping	real	integer	complex	float	1157	293
27	**data_carriers_map**	- data carriers mapping	complex	float	complex	float	293	385
28	**data_IFFT**	- data IFFT	complex	float	complex	float	385	385
29	**data_cyclicprefix**	- data cyclic prefix	complex	float	complex	float	385	480
30	**data_map**	- receive time data samples - send time data samples to PPDU	complex	float	complex	float	480	480
31	**PPDU_map**	- receive time data samples - form PPDU packet for transmission	complex	float	complex	float	480	880

B. RECEIVER

	IEEE 802.11a Receiver components sequential flow						Example (802.11a Annex G)	
	Components	**Description**	**Data In**	**Type In**	**Data Out**	**Type Out**	**Data size in**	**Data size out**
1	**Rx_data**	- received digitised data stream	-	-	complex	float	-	continuous
2	**PPDU_rx**	- extract the required digitised PPDU stream - removed preamble from PPDU - send stream for header removal	complex	float	complex	float	continuous	560
3	**Header_rx**	- removed header from PPDU - send header for processing	complex	float	complex	float	560	80
4	**SIG_cyclicprefix_rem**	- SIG cyclic prefix removal	complex	float	complex	float	80	64
5	**SIG_FFT**	- SIG FFT	complex	float	complex	float	64	64
6	**SIG_carriers_demap**	- SIG carriers demapping	complex	float	complex	float	64	48
7	**SIG_BPSK_demod**	- SIG BPSK demodulation	complex	float	real	integer	48	48
8	**SIG_deinterleaver**	- SIG deinterleaving	real	integer	real	integer	48	48
9	**SIG_conv_dec**	- SIG convolution decoding	real	integer	real	integer	48	24
10	**Header_rx**	- extract RATE & LENGTH from SIG - send received data for processing	real	integer	complex	float	24	485
11	**data_rx**	- receive and send raw data for processing	complex	float	complex	float	485	485
12	**data_cyclicprefix_rem**	- data cyclic prefix removal	complex	float	complex	float	485	389
13	**data_FFT**	- data FFT	complex	float	complex	float	389	389
14	**data_carriers_demap**	- data carriers demapping	complex	float	complex	float	389	293
15	**data_demod_map**	- data demodulation mapping	complex	float	real	integer	293	1157
16	**data_deinterleaver**	- data deinterleaving	real	integer	real	integer	1157	1157

	IEEE 802.11a Receiver components sequential flow						Example (802.11a Annex G)	
	Components	**Description**	**Data In**	**Type In**	**Data Out**	**Type Out**	**Data size in**	**Data size out**
17	**data_conv_dec**	- data de-puncturing - data convolution decoding	real	integer	real	integer	1157	869
18	**data_descrambler**	- descrambler the raw data	real	integer	real	integer	869	864
19	**data_rx**	- receive and send PPDU data to MAC layer	real	integer	real	integer	864	801
20	**data_rx_PSDU_display**	- display the PSDU received bits	real	integer	-	-	801	

LIST OF REFERENCES

[1] "Definition of Iterative and Incremental development, Wikipedia". http://en.wikipedia.org/wiki/Iterative_and_incremental_development last accessed on Oct 2006.

[2] J. H. Reed, "Software Radio: A Modern Approach to Radio Engineering", 1st ed. New Jersey: Prentice Hall, 2002.

[3] IEEE Std 802.11a-1999 (Revision 2003), "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications, high-speed physical layer in the 5 GHz Band".

[4] Matthew S. Gast, "802.11 Wireless Networks – The Definitive Guide", Chapter 11: 802.11a - 5GHz OFDM PHY, O'Reilly, 2002.

[5] Jacob A. DePriest, "A Practical Approach to Rapid Prototyping of SCA Waveforms" thesis, Virginia Polytechnic Institute and State University, 2006.

[6] Hiroshi Harada & Ramjee Prasad, "Simulation and software radio for mobile communications", Chapter 4: OFDM Transmission Technology, Boston: Artech House, 2002.

[7] C. Britton Rorabaugh, "Simulating Wireless Communication Systems: Practical Models in C++", New Jersey: Prentice Hall PTR, 2004.

[8] S.M. Shajedul Hasan, "A Software Defined Radio Receiver based on USRP and OSSIE Framework" Course Project - ECE 5674: Software Radios, Virginia Polytechnic Institute and State University, 2006.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Jeffrey B. Knorr
   Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

4. Asstistant Professor Frank Kragh, Code EC/Kh
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

5. Professor R. Clark Robertson, Code EC/Rc
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

6. Professor Carl Deitrich
   Virginia Polytechnic Institute and State University
   Blacksburg, VA

7. Professor Jeffrey H. Reed
   Virginia Polytechnic Institute and State University
   Blacksburg, VA

8. Professor Charles W. Bostian
   Virginia Polytechnic Institute and State University
   Blacksburg, VA

9. Mr. Howard Pace
   JTRS, JPEO
   San Diego, CA

10. Dr. Richard North
    JTRS, JPEO
    San Diego, CA

11. Ms. Donna Miller
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA

12. Mr. Nathan Beltz
SPAWAR Systems Center
San Diego, CA